FORMALIZATION OF THE STATIC SEMANTICS OF CONTRACTS USING ATTRIBUTE GRAMMARS

par

Qun Wu

mémoire présenté au département de mathématiques et d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, juin 1996

ISBN 0-612-15496-3

ABSTRACT

The work presented in this thesis is about the Contracts language. It is often the case that groups of related objects cooperate to perform some tasks or maintain some invariants within an object-oriented system. Helm's Contracts language is a technique for specifying this kind of inter-object cooperations in terms of contracts. A contract in Contracts specifies the cooperations in terms of participating objects, obligations of the participants, dependencies between the participants, and the instantiation of the participants. Furthermore, a contract contains invariants which participants cooperate to maintain.

The results of this thesis are a formal specification and a partial implementation of the static semantics of the Contracts language. The syntax used for contracts comes from Holland's work and the formalization is done using attribute grammars. Attribute grammars give the formal definition of all context-free and context-sensitive language properties. They define each language construct only by its *immediate environment* and minimize the interconnection between the different parts of the language, which makes the definition easier to understand and more concise. The context-free part of the grammar has been implemented using Lex and Yacc.

Sommaire

Introduction

Le travail présenté dans ce mémoire concerne le langage Contracts. Les systèmes orientés objet consistent en des groupes d'objets reliés qui coopèrent afin de réaliser certaines tâches ou de maintenir certains invariants. Le langage Contracts de Helm est une technique de spécification de coopérations interobjets [Helm 90]. Un contrat, dans Contracts, spécifie les coopérations en termes d'objets participants, d'obligations des participants, de dépendances entre participants et de l'instanciation de ceux-ci. De plus, un contrat contient des invariants que les participants coopèrent à maintenir.

Les résultats de ce mémoire sont la formalisation de la sémantique statique du langage Contracts en utilisant la notation de grammaires d'attributs [Wait 85], selon la syntaxe publiée dans [Holl 92], et l'implémentation de l'analyse syntaxique et lexicale de Contracts en utilisant Lex et Yacc.

Techniques orientées objet et contrats

Le but principal des techniques orientées objet est d'améliorer la fiabilité des systèmes logiciels. La théorie des contrats intervient comme une approche plus systématique pour l'obtention et la garantie de la fiabilité. Les groupes d'objets coopérant afin d'accomplir des tâches ou de maintenir certains invariants, qui sont appelés compositions comportementales, sont un élément important des systèmes orientés objet. Les contrats visent à formaliser les collaborations et les relations comportementales entre objets. Selon la définition initiale du terme, un contrat est un ensemble de responsabilités interreliées

définies par une classe. Il décrit les façons dont dispose un client donné pour interagir avec un serveur. Un contrat est une liste de requêtes qu'un client peut formuler à un serveur. Le client et le serveur remplissent le contrat en exécutant uniquement les requêtes que le contrat spécifie et en répondant adéquatement à ces requêtes séparément. Les travaux scientifiques récents reconnaissent l'importance de telles coopérations. Ils font référence au fait que les comportements inter-objets peuvent être exprimés sous la forme de graphes de responsabilités [Wirf 89] et de collaborations [Wirf 90]. Une responsabilité est quelque chose qu'un objet réalise pour d'autres objets ; c'est la base de la détermination des contrats supportés par une classe. Un contrat constitue un ensemble cohésif de responsabilités sur lequel un client peut compter. Un graphe de collaborations est un outil d'aide à l'analyse des voies de communication et à l'identification de sous-systèmes potentiels. Il montre graphiquement la collaboration entre classes et sous-systèmes. Il représente les classes, les contrats et les collaborations. Il montre également les relations super-classe et sous-classe. Dans les graphes de collaborations, une super-classe représente les contrats supportés par toutes ses sous-classes. Mais l'inférence des dépendances comportementales qu'elles impliquent est difficile et celles-ci causent des problèmes subséquents dans la conception, la compréhension et la réutilisation des logiciels orientés objet.

Il y a plusieurs usages différents du mot *contrat* dans le domaine de l'orienté objet. Chaque usage est basé sur le principe de l'encapsulation et des conséquences de l'application de ce principe. Cependant, ces usages du mot *contrat* diffèrent principalement dans l'étendue de leur définition en termes de nombre d'objets participants et des méthodes que chaque participant supporte. Par exemple, le concept de contrat de Meyer ne fait intervenir que deux classes (la classe appelante, qui est le client, et la classe appelée, qui est le fournisseur) et une méthode [Meye 92]. Le client invoque la méthode implémentée par le fournisseur. Le contrat est exprimé en utilisant deux sortes d'assertions. La première espèce d'assertions exprime les obligations et les bénéfices du contrat, qui sont appelés respectivement préconditions et postconditions. Les préconditions expriment des spécifications qui font

appel à une routine à satisfaire pour être correctes. Les postconditions expriment les propriétés qui doivent être garanties après le retour de l'exécution de la routine. La deuxième espèce d'assertions est constituée par les invariants de classe, qui sont les propriétés qui s'appliquent à toutes les instances de la classe, transcendant les routines particulières. Meyer a introduit le concept de contrat dans le langage Eiffel, mais à un niveau plutôt bas d'abstraction et de formalisme.

Le concept de contrat de Helm généralise le concept initial de contrat pour deux entités aux relations multiobjets. Il s'attache à l'interaction entre objets coopérants. Il supporte la découverte, la compréhension et la représentation de l'interaction entre objets. Il fournit un vocabulaire pour la conception orientée interaction. Par comparaison avec d'autres notions de contrat, la contribution de Helm est la généralisation des dépendances aux multiobjets, la saisie des dépendances comportementales entre objets coopérants et un formalisme pour l'abstraction.

Les contrats de Helm spécifient les compositions comportementales sous la forme d'objets participants, d'obligations contractuelles de chaque participant, de préconditions sur les participants requises pour établir un contrat et d'invariants que les participants doivent maintenir. Les obligations contractuelles sont des obligations de type et des obligations causales. Les obligations de type spécifient certaines variables et interfaces externes qui sont supportées par les participants. Les obligations causales spécifient des séquences ordonnées d'actions que doivent exécuter les participants. Les obligations causales sont l'élément essentiel des compositions comportementales. Par l'intermédiaire des obligations causales, les contrats rendent explicites les dépendances comportementales entre les participants d'un contrat. Il y a deux opérations sur les contrats : l'inclusion de contrat et le raffinement de contrat. Le comportement des participants d'un contrat peut être spécialisé ou étendu grâce à ces deux opérations. Le raffinement permet la spécialisation des obligations contractuelles et des invariants des contrats. Les contrats peuvent être

raffinés en spécialisant les types de participants ou en dérivant un nouvel invariant qui implique l'ancien. Les obligations raffinées se basent sur celles du contrat raffiné. Toutes les autres obligations du nouveau contrat sont héritées de l'ancien contrat.

L'inclusion permet aux contrats d'être formés de sous-contrats plus simples. La participation à un sous-contrat impose des obligations additionnelles aux participants au-delà et en plus de celles définies dans le contrat. Elles sont impliquées par les sous-contrats inclus plutôt que réécrites. Les mécanismes d'inclusion et de raffinement peuvent être combinés si nécessaire. Tous deux fournissent le moyen d'exprimer des comportements complexes sous la forme de comportements plus simples et permettent aux contrats de créer et de réutiliser des abstractions grossières basées sur les comportements. Les contrats utilisent des déclarations de conformité pour définir comment l'implémentation d'une classe particulière, et donc de ses instances, rencontre les obligations d'un type de participants. Les contrats sont définis dans un langage de haut niveau, qui permet une description sommaire des comportements en termes de séquences ordonnées d'actions à effectuer et de conditions à rendre vraies.

Plusieurs chercheurs ont introduit et utilisé le concept de contrat de Helm dans leurs travaux. Buhr et Casselman ont développé une vision unifiée des architectures logicielles en termes de contrats, de rôles et de "timethreads" [Buhr 92]. Aussi bien les architectures câblées que les architectures non connectées se concentrent sur l'expression des relations de coopération entre composants et sur les interactions des composants entre eux. Ils ont affirmé que l'élaboration d'architectures connectées en rôles et en contrats aide à montrer les aspects distincts, simples et séparés de la conception. Voir les architectures non connectées en termes de contrats fournit un niveau plus élevé d'abstraction dans la conception. Lajoie et Keller ont développé un modèle multicouche pour la réutilisation d'un modèle [Lajo 94]. Les contrats sont introduits dans leurs travaux comme une technique de description pour les spécifications de haut niveau de comportements d'objets. Les travaux

de Holland sur les contrats portent à la fois sur la sémantique statique et sur la sémantique dynamique du langage [Holl 92] [Holl 93]. La sémantique statique de chaque construction grammaticale est décrite sous la forme d'une fonction de validité et exprime des contraintes additionnelles qui ne peuvent être exprimées par la grammaire. La sémantique dynamique de chaque construction grammaticale est définie sous la forme d'une fonction :

$$M_T : T \rightarrow (Etats \rightarrow Etats).$$

Ceci signifie que chaque spécimen de la construction T est affecté à une fonction (possiblement partielle) de Etats dans Etats. En utilisant ce style de spécification, Holland a construit un modèle séquentiel de calcul.

Grammaires d'attributs

Notre formalisation de la sémantique statique du langage Contracts est faite en utilisant la notation de grammaires d'attributs [Wait 85]. Les arbres de structure sont utilisés pour illustrer l'application des règles de syntaxe des contrats à l'analyse d'un texte dans ce langage. Chaque noeud de l'arbre de structure correspond à l'utilisation d'une règle et est doté d'attributs décrivant les propriétés de cette règle. L'attribution de chaque noeud dans l'arbre structural collecte l'information sur son environnement. Les grammaires d'attributs sont utilisées pour représenter les règles de l'attribution. Elles consistent en la définition formelle de toutes les propriétés de langage indépendantes et dépendantes d'un contexte.

Une grammaire d'attributs associe un ensemble fini A(x) d'attributs à chaque symbole x appartenant au vocabulaire de la grammaire. Elle peut être definie par des fonctions associées à chaque production dans la grammaire. Chaque attribut représente une propriété contextuelle d'un symbole et peut être synthétisé ou hérité. Les attributs synthétisés représentent des propriétés du point de vue du sous-arbre dérivé du symbole dans l'arbre de structure. Les attributs hérités résultent de la considération de l'environnement. Les

attributs synthétisés sont suffisants pour définir toute fonction d'un arbre de dérivation. L'inclusion d'attributs hérités peut résulter en d'importantes simplifications. D'autres techniques pour la définition sémantique formelle de langages de programmation incluent l'algorithme markovien de croissance de Bakker et le λ-calcul de Landin [Land 64] [Land65] [Land 66]. La différence la plus frappante entre ces méthodes et les grammaires d'attributs est que ces techniques sont des processus qui sont définis par des programmes comme un tout, et ce, d'une façon plutôt compliquée. On doit comprendre en entier un compilateur d'un langage avant de pouvoir comprendre la définition de ce langage. La grammaire d'attributs définit chaque construction du langage seulement par son environnement immédiat, en minimisant les interconnections entre les différentes parties du langage. Cette localisation et ce partitionnement des règles sémantiques tendent à rendre la définition plus facile à comprendre et plus concise. Les grammaires d'attributs donnent la définition formelle de toutes les propriétés du langage, non contextuelles et contextuelles, et constituent aussi une spécification formelle de l'analyse sémantique.

Spécification formelle de la sémantique statique des contrats avec les grammaires d'attributs

Dans notre travail, nous adoptons les conventions, fonctions et notations de Waite et Goos pour formuler la grammaire d'attributs pour le langage Contracts [Wait 85]. Chaque production de Contracts est exprimée en utilisant la notation EBNF et commence avec le mot-clé **rule**. La partie attributs commence avec le mot-clé **attribution** et une condition peut suivre le mot-clé **condition**.

Cette partie de notre travail définit les propriétés sémantiques statiques du langage Contracts au moyen d' une grammaire d'attributs. Elle consiste en deux parties principales. La première partie traite de l'introduction des outils, incluant les définitions des structures

pour la présentation des identificateurs et des types, et de toutes les fonctions dans la suite. La seconde partie consiste en toutes les règles de Contracts. Cette partie détermine les propriétés statiques de Contracts et introduit une structure de spécification de Contracts: comment déclarer les participants dans un contrat, comment définir les obligations de chaque participant (incluant les variables et les méthodes supportées), comment définir un contrat incluant ou raffinant quelques contrats existants et comment exprimer les invariants d'un contrat et de son instanciation. Dans cette partie, les principaux aspects de la sémantique statique de Contracts sont discutés par l'intermédiaire de la grammaire d'attributs, comme l'étendue d'un identificateur et les types et les différentes relations entre eux (par exemple, équivalence de types, compatibilité de types).

Implémentation d'un analyseur syntaxique

Notre travail inclut aussi l'implémentation de l'analyse syntaxique et lexicale de Contracts en utilisant Lex et Yacc. L'analyse lexicale traite de la division des intrants en unités significatives qui sont appelées "jetons". La prétraduction traite de la découverte des relations existant entre les unités. Lex est un outil de construction d'analyse lexicale [Levi 92]. Il génère une fonction C qui peut identifier des jetons en prenant un ensemble de descriptions, appelé spécification Lex. Yacc est un outil pour la génération d'un analyseur qui reconnaît les phases valides de la grammaire [Levi 92]. Il prend comme intrants une série de règles, qui est appelée une grammaire Yacc.

Notre spécification Lex pour Contracts consiste en trois parties : la définition, les règles et les sections de sous-routines des utilisateurs. La section de définition inclut un fichier généré par Yacc et contient toutes les définitions de jetons. La section des règles identifie les différents jetons utilisés par Contracts. Ils comprennent les mots réservés et quelques autres jetons de connexion tels que les identificateurs, les nombres et les opérateurs. La

section des sous-routines des utilisateurs inclut les routines de traitement des erreurs et la routine principale. La routine principale ouvre un fichier nommé sur la ligne de commande et appelle la prétraducteur. La valeur retournée rapporte le fait que la prétraduction a réussi ou échoué.

Notre grammaire Yacc pour Contracts comprend deux parties : la section des définitions et celle des règles. La section des définitions déclare les symboles terminaux que l'analyse lexicale passe au prétraducteur. Tous les symboles utilisés comme jetons doivent être définis explicitement dans la section des définitions. La section des règles comprend toutes les règles de grammaire, comme les règles de déclaration de chaque participant, l'inclusion, le raffinement des contrats existants de chaque participant, les invariants et l'instanciation d'un contrat.

Lorsqu'un scanner Lex et un analyseur Yacc sont utilisés ensemble, l'analyseur est la routine de haut niveau. L'analyseur appelle le scanner lorsqu'il a besoin d'un jeton depuis les intrants. Le scanner et l'analyseur doivent se mettre d'accord sur ce que sont les codes des jetons. Ceci est résolu en laissant Yacc définir les codes des jetons dans un fichier ; l'analyseur lexical utilise ces définitions numériques des jetons en incluant le fichier.

Nous avons utilisé notre analyseur pour vérifier trois exemples tirés de [Helm90]. Ces trois exemples sont Contracts DepthFirst, Dft-Connected et Dft-cycle. Nous avons aussi utilisé notre analyseur pour vérifier un exemple présenté dans ce mémoire.

ACKNOWLEDGEMENTS

First and foremost, I wish to extend special thanks to my research director Dr Michel Barbeau. He introduced me to the fascinating world of my research work. He gave me a lot of help in my courses, my research work, and the writing of this thesis. I will always bear this in mind.

My acknowledgements also to all the professors and students in our department.

I would also like to acknowledge the financial support of the ministère de l'Industrie, du Commerce, de la Science et de la Technologie du Québec.

CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

This work is about the formalization and partial implementation of the static semantics of the Contracts language. The syntax used for Contracts comes from [Holl 92], which is more mature and slightly different from the original syntax published in [Helm 90]. The static semantics has been formalized using attribute grammars. The implementation has been done using Lex and Yacc.

In the sequel, we introduce Contracts and attribute grammars, we state the results, and outline the contents of this thesis.

1.1 Introduction to Contracts

Improving the reliability of software systems is the main goal of object-oriented techniques. Defensive programming is an often emphasized approach when discussing reliability [Wirf 89]. This technique suggests that routines should be as general as possible. However, the blind and often redundant checking in this technique causes much of the complexity and affects the quality and reliability of systems.

The concept of contract pretends to be a more systematic approach for obtaining and guaranteeing reliability. Initially, a contract is a class related concept [Wirf 89]. A contract describes how a given client can interact with a server. A contract is a list of requests that a client can make to a server. Both cooperate to fulfil the contract in the following way. The client makes only those requests that the contract specifies and the server responds appropriately to those requests. Helm's notion of contract generalizes the initial concept

of contract, for two entities, to multiobject relations. It aims at explicitly specifying the important behavioral dependencies caused by object interactions. It focuses on the interaction of cooperating objects and supports the discovery, understanding, and representation of object interactions.

Understanding and representation of object interactions are also important with respect to reuse. Reuse is the cornerstone of the object-oriented technology. An abstract design can be expressed as a group of classes. Reusing this abstract design means both reusing the classes in the group and the relationships between them.

Helm's Contracts defines behavioral dependencies, called object interactions, within a group of objects which cooperate to accomplish some tasks or maintain some invariants. Each contract specifies the participating objects in interaction and also the function of each object that contributes to the interactions. Apart from the explicit representation of object interactions, Contracts also provides a module-like construct to structure and organize object-oriented implementations. Contracts breaks up the class behaviors and packages the pieces in cohesive contracts. This overcomes the problems caused by some other existing module architectures consisting of either a small number of very large modules or a spaghetti-like structure of import/export module relationships.

Contract refinement and contract inclusion are two important features of the Contracts language. They provide two distinct means to express complex behaviors by simpler ones. Through these techniques, components defined as contracts can be customized through type substitution and method overriding. Large-grained abstractions based on object interactions can be created and reused.

The creation of object-interactions is performed by an instantiation statement. An instantiation statement identifies objects as participants in a desired contract and specifies the methods which are used to establish the contract. This is different from the previous

programming language constructs in which object interactions are created by class constructors or certain sequences of method calls. Contract instantiation statements bring the benefits of I) explicitly expressing the behavioral composition and the objects involved in the interaction, ii) providing better and easier understanding of the application architecture, and iii) facilitating the replacement of objects by functionally equivalent objects, when a change is needed.

Finally, a conformance statement establishes the fact that the behavior defined for a class conforms to the behavior required by a contract. It is used to determine the legal classes of objects allowed to participate in a particular contract. It can also be considered to be a validation step, ensuring that an individual has the authority and ability to participate in a contract. Contract conformance and contract instantiation are two steps to establish a contract between a group of objects in an object-oriented system.

Comparing Contracts with other techniques for specifying inter-object relationships, the contributions of Helm's contract are the following: generalizing the dependencies to multiple objects, capturing the behavioral dependencies between cooperating objects, and providing a formalism for abstraction. As the author claimed in [Helm 90], an initial shift away from class-based design to one based on Contracts greatly facilitates the early identification, abstraction, and subsequent reuse of patterns of interactions between objects.

1.2 Introducing Attribute Grammars

In this thesis, the formalization of the static semantics of the Contracts language is done using the notation of attribute grammars. Structure trees can be used to illustrate the use of the syntax rules of a language for analysing a text in that language. Each node of the structure tree corresponds to the use of a rule of the language and can be *decorated* with attributes describing the properties of that rule. The attribution of each node in the structure

tree collects information about the environment. Rules of attribution can be represented by attribute grammars which consist of the formal definition of all context-free and context-sensitive language properties.

An attribute grammar is based on a context-free grammar $G = (V, N, S, \Phi)$. For each symbol $x \in V$, there is a finite set $A(x)$ of attributes which represents a context-sensitive property of x. Each attribute can be either a synthesized attribute or an inherited attribute and is defined by functions associated with each production in the grammar.

The initial use of attribute grammars is to define semantics by associating synthesized attributes with each nonterminal symbol and semantics rules with each production. Each nonterminal symbol is given exactly one attribute. Although synthesized attributes alone are sufficient, the inherited attributes lead to important simplifications. Inherited attributes facilitate the definition of a *block structure,* which is a common aspect of programming languages. Generally, inherited attributes are useful when a part of the meaning of some construction is determined by the context in which that construction appears.

The semantic rules are said to be well-defined if they are formulated in such a way that all attributes can always be defined at all nodes in any derivation tree. It is important to decide whether or not a given grammar has well-defined semantic rules, since there may be infinitely many derivation trees. The definition and theory of "well-definedness" can be found in [Wait 85], and an algorithm for testing "well-definedness" can be found in [Knut 68].

Some other techniques for the formal semantic definition of programming languages include Bakker's growing Markovian algorithm [Bakk 67] and Landin's λ-calculus. Both have been used [Land 64] [Land 65] [Land 66] for the definition of ALGOL 60. The most striking difference between these methods and attribute grammars is that the former are processes defined on programs as a whole in a rather intricate manner. One must

understand an entire compiler for the language before one can understand the definition of the language. The attribute grammar minimizes the interconnections between the different parts of the language by defining each language construct only with its *immediate environment*. Because of the localization and partitioning of the semantic rules, it makes the definition easier to understand and more concise. Attribute grammars give the formal definition of all context-free and context-sensitive language properties and also constitute a formal specification of the semantics analysis.

1.3 Results and Outline

The main result of this thesis is a formal specification, using attribute grammars, of the static semantics of Contracts, according to the syntax published in [Holl 92].

This thesis also resulted in an implementation, using Lex and Yacc, of the context-free part of this attribute grammar.

This thesis is organized as follows. We first review in Chapter 2 the work related to Contracts. Chapter 3 gives the formal specification of the syntax and static semantics of the Contracts language and the intuitive description of the dynamic semantics of each construct. Chapter 4 gives the results of the implementation of the context-free part of the attribute grammar of Contracts using Lex and Yacc. Chapter 5 gives an example that illustrates the application of Contracts to the specification of object interactions in a mine drainage control system problem [Barb 94]. Finally, Chapter 6 gives the conclusions.

Chapter 2

Related Work

2.1 Introduction

This chapter is a survey of work related to ours. It includes the notion of Helm's Contracts which is the basis of our work, Holland's work based on Helm's Contracts with some differences and additions, Buhr's work about Contracts, and other works related to Contracts.

As there are many different uses of the word "contract", this section compares and contrasts the different uses of the term contract in the object-oriented field. Each notion of contract discussed below is based on the principle of encapsulation and the consequences of applying this principle. The notions differ mainly in the scope of their definitions in terms of the number of objects participating in contracts and the methods each participant supports. This chapter also reviews some other techniques for the specification of inter-object behavior, e.g., collaboration graphs and responsibilities. Finally, for the reason that our intention is to apply Contracts as a language of formal specification, formal specification of requirements in an object-oriented context is briefly discussed, that is, Hayes and Coleman's work.

2.2 The Notion of Helm's Contracts

Helm's Contracts is a technique for specifying the obligations on participating objects within a behavioral composition. Such a behavioral composition involves groups of related objects

cooperating to perform a task and maintain invariants [Helm 90]. A contract specifies a behavioral composition in terms of: I) the objects participating in the composition, that is, the participants; ii) the obligations of the participants; iii) the dependencies between the participants; and iv) the instantiation of the contract. Participant obligations include: I) support of a certain interface described as typed variables and messages, it is called type obligations, ii) performance of sequences of actions and satisfaction of conditions in response to messages, and iii) satisfaction of preconditions at the instantiation of the contract and maintenance of invariants during the contract. The later two aspects are called causal obligations. Causal obligations capture the behavioral dependencies between objects.

A contract contains invariants that participants cooperate to maintain and it also defines what actions should be initiated to resatisfy the invariants when they are violated. A contract includes preconditions on participants to establish the contract and the methods which instantiate the contract.

Contracts are defined in a high-level language, which allows abstract description of behavior in terms of ordered sequences of actions to be performed and conditions to be made true. The language supports the following basic actions in the original syntax in [Helm 90]: sending a message, denoted by P→M and the setting of an instance variable v, denoted by Δv. The ordering of actions can be explicitly given by the operator " ; ". The language also provides the construct $< ov : c : e >$ for the repetition of an expression "e" separated by the operator "o" for all variables "v" that satisfy "c". For example, $< ||v: v \in Views: v \rightarrow Update() >$ means:

$v1 \rightarrow Update() \,||\, v2 \rightarrow Update() \,||\, v3 \rightarrow Update()$ ... if v1, v2, v3,...∈Views.

Conditions which participants are obliged to make true appear in curly brackets { ... } and are expressed as logical formulae over the signatures (the names of the sorts and operators, together with the definition of domains and ranges) of participants.

There are facilities for refinement of contracts and inclusion of subcontracts in a given contract. Refinement makes possible specialization of a contract by the overriding or addition of participant obligations. Inclusion allows decomposition of a contract into simpler subcontracts. Both facilities are orthogonal and promote reuse of contracts.

Contracts make abstraction of actual classes of objects (e.g., C++ classes) describing the implementation of the participants. Indeed, several actual classes can meet the obligations of a participant. Conversely, a class can meet the obligations of several participants. The precise mapping of participants to actual classes is described by means of a conformance declaration. Syntactical details and an informal definition are presented in [Helm 90].

2.3 Holland's Work about Contracts

Holland's work about Contracts encompasses both static semantics and dynamic semantics of the language [Holl 92][Holl 93]. Both aspects are discussed hereafter.

Static Semantics

In Holland's work, the static semantics of each grammar construct is described in terms of a validity function:

$$V_T : T \rightarrow B$$

where T is an abstract syntax construct defined by the grammar and B={true, false}. A static semantics usually expresses additional constraints which cannot be expressed by the grammar. For example, the semantics of a procedure call (denoted as procCall in the abstract syntax) includes a constraint saying that every procedure call must refer to a procedure defined in the program. It is defined as follows:

$$\text{Vproccall} : \text{ProcCall} \times \text{ProcEnv} \rightarrow \text{B}$$

$$\text{Vproccall}[[\text{iCall, ProcEnv}]] \equiv \{\text{iCall.procName} \in \text{dom ProcEnv}\}.$$

In Holland's work, the validity function V_T is not provided for every construct of Contracts. Our work about static semantics of contract is based on an attribute grammar and we define an attribute grammar rule for every construct of Contracts.

Dynamic Semantics

In Holland's work, the dynamic semantics for each grammar construct is defined in terms of a function:

$$M_T : T \rightarrow (\text{State} \rightarrow \text{State})$$

It means that each specimen of the construct T is mapped to a (possibly partial) function taking one element of *State* to another. By using this denotational semantics style method, Holland constructs a sequential model of computation.

2.4 Buhr and Casselman's Work About Contract

Buhr and Casselman's work [Buhr 92] aims to develop a unified view of software architectures in terms of contracts, roles, and timethreads. The main idea of contract in their paper is not only based on Helm's contract, but also comes from other terms for describing inter-object relations such as mechanisms and cooperations.

Buhr and Casselman developed the concepts of wired and wireless architectures. *Architectures* focus on the collaboration relationships between components and how these components interact with each other to accomplish system tasks. Two kinds of unifying models of software architectures are wired architectures and wireless architectures. Wired

architectures are static one. They only permit communication between the components which are wired together. Wireless architectures are relatively dynamic. They permit communication between any two components at any time. Wireless architectures are proposed to be suitable for data-centric design and wired architectures for behavior-centric design.

Contracts are the rules or protocols which the components must obey in accomplishing system tasks. They specify how groups of object cooperate with each other. Objects participate in contracts by playing roles. A role defines a type obligation, including a set of methods with their type signatures (the names of the sorts and operators, together with the definition of domains and ranges), and a causal obligation. Contracts can be viewed as entities of the design. Role playing is static in wired architectures and dynamic in wireless architecture.

In wired architectures, timethreads are proposed to express the major patterns of contracts. Contracts can be expressed by one or more timethreads. A timethread starts at some point of stimulus, go through all design elements which are activated by the stimulus sequentially, and ends at some point of the completion of all activities caused by the stimulus.

Contracts help us to understand a framework. Factoring wired architectures into roles and contracts help to show the distinct, simple, and separable aspects of a design. Viewing wireless architectures in terms of contracts provides a high level of abstraction of design. Representation based on contracts is useful for documenting object-oriented frameworks and also provide a tool for the design of them.

2.5 Lajoie and Keller's work about Contracts

Software reuse is a means of improving the practice of software engineering. Object-oriented frameworks improve design-level reuse. A framework is a collection of abstract and concrete classes and the interfaces between them. A framework is the design for a subsystem.

In a framework, there are collaborations between the objects in classes. The interaction among objects, however, cannot be expressed very well by frameworks. Micro-architectures have been introduced to codify design knowledge about object collaborations.

Lajoie and Keller [Lajo 94] developed a multi-layered model for framework documentation and reuse. Description techniques for both micro-architectures [Gamm 93] and frameworks adopted and refined in their work include: design patterns [Alex 79], Contracts [Helm 90], and Motifs [John 92]. Design patterns is a technique for describing micro-architectures at a very high level. They express how components interrelate and give a high-level representation for properly capturing and describing design experience. They can be described in an informal template-based manner. The Contracts technique is adopted as an intermediate representation between a micro-architecture and its corresponding design pattern. The contract definition in this work comes directly from Helm et al. and is also used as a construct for explicitly specifying interactions among groups of objects. The main adjustment to Contracts in this work is that: micro-architectures are allowed to be developed directly and from a guiding design pattern. Contracts are used to describe high-level specifications of object behavior, with emphasis on minimum details required to express participant interactions. Contrary to Helm's contracts, the object interactions are first class entities in the design space. This paradigm supports interaction-oriented design. In this paradigm, design is a two-step process, firstly, defining behavioral compositions by contracts, secondly, factoring contracts into classes definitions and hierarchies via the

contracts conformance declarations. Motifs are used to describe the general, high-level aspects of the framework rather than the design details of the micro-architectures. All motifs have the same form which begins with a title followed by a detailed discussion of how the situation may be adopted and ends with references to other related motifs as well as to relevant design patterns or contracts. The integrated description techniques proposed by the authors in their framework is to decrease a class framework's learning curve and consequently increase their reusability.

In a word, design patterns and contracts are introduced as new techniques for describing micro-architectures at a very high level and an intermediate level respectively. Motifs are adapted to the description of the abstraction levels of framework design.

2.6 Other Tools for Understanding Object Interactions

Explicitly specifying inter-object relationships is not entirely new. Many recent publications recognize the importance of inter-object behavior, which can be expressed in terms of both conceptual and graphical tools; e.g., responsibilities [Wirf 89] and collaboration graphs [Wirf 90], mechanisms, and views. The first two terms are reviewed in this section.

A contract is a set of related responsibilities defined by a class. Responsibilities are the basis for determining the contract supported by a class. A responsibility is something one object does for other objects by either performing some action or responding with some information. It is the central idea for responsibility-driven design which focuses on what is the object responsible for and what information this object shares. A contract constitutes a cohesive set of responsibilities.

A collaboration graph is used to display graphically the collaborations between classes and

subsystems. A subsystem is a set of classes collaborating to fulfil a set of responsibilities. A collaboration graph simplifies a design by identifying subsystems within a complex large application and it clearly represents classes, contracts, collaborations, and superclass-subclass relationships. A superclass represents the contracts supported by all of its subclasses.

2.7 Meyer's Work

As the complexity of applications increases, it is important to improve the ability of software to be reused, refined, tested, maintained, and extended. Abstraction is an effective tool to improve these abilities of software. Many types of abstraction can be used. Encapsulation is a key form of abstraction by which complexity can be managed. Since programming in an object-oriented language does not ensure that the complexity of an application will be well encapsulated, good programming techniques need to be applied to improve encapsulation.

Meyer proposed the notion of "programming by contract". A notion of contract is used to specify the relationship between the caller and the called routine of a class with associated benefits and obligations [Meye 92]. It involves precisely two classes and one method. One of the classes (the caller class, called the client) invokes a method implemented by the other (the callee class, called supplier). The contract can be expressed using two kinds of assertions. The first kind of assertions are pre and postconditions expressed using the keywords *require* and *ensure*. The second kind are class invariants. The obligations and benefits of the contract are specified by preconditions and postconditions. The preconditions express requirements that must be satisfied for any call of a routine. The postconditions express properties that must be held after the return of the execution of the call. A class invariant is a property that applies to all instances of a class, transcending

particular routines. It defines constraints among class instances and methods. Meyer introduced its notion of contract in the Eiffel language, but at a rather low level of abstraction and formalism.

In contrast, Helm's notion of contract is used to specify the behavioral composition of several cooperating participants (may be more than two) and the invariants are global to all the participants. The main difference is that Helm's contract generalizes the contract notion to multiobject dependencies.

Let us consider the example of Fig 1. The left part of Fig.1 defines the *put-child* contract using Meyer's notion of contract. The contract is related to a class *binary-tree* describing a routine *put-child* for adding a new child to a tree node, denoted as *current*. The require part of the contract requires that the child be accessible through a reference, which must be attached to an existing node object. The ensure part of the contract expresses that, in return of the execution of the call, the tree is updated. That is, the current node has one more child than before and *current* is the parent of the new node.

The invariant of this contract states that node *current* is the parent of both the left and right children of node *current*, if these children exist. This contract shows that the client class gets the benefits of an updated tree where the *current* node has one more child than before. *New* has *current* as its parent now, but the obligations are that *new* is used as an argument and it references to an existing node. The supplier class of this contract gets the benefits that: no need to do anything if the argument is attached to a void object and it must guarantee that insertion of a new node as required.

```
/* Meyer contract*/
Class binary-tree[T]
    [ - attribute declarations

     - routine declarations

    put-child(new:Node)
    require new/=void
    do
    -insertion algorithm
    ensure new.parent=current
    child-count=old child-count+1
    ...]


    invariant
    left/=void ⇒
            (left.parent=current)
    right/=void ⇒
            (right.parent=current)
```

```
/* Helm contract*/
Contract UndirectedGraph
participants
    graph:Graph;
    vertices:Set(Vertex);

Graph supports [
    vertices:Set(Vertex);
    insert(v:Vertex):void;
            ...]


Vertex supports [
    neighbours:Set(Vertex);
            ...]

        invariant
          graph.vertices=vertices∧
          <∀ v∈vertices:
          v.neighbours⊆vertices>∧
          <∀ v∈vertices:
              <∀ v2∈v.neighbours:
               v∈v2.neighbours>>

end contract
```

Figure 1 : Examples of Meyer's approach (left side) and Helm's approach (right side)

The right part of Fig.1 defines the *UndirectedGraph* contract by using Helm's notion of contract. It consists of two kinds of participants: a graph object and a set of vertices. The contractual obligations of the participants are expressed, with the "support" clauses and the invariant ensures that the participating objects do indeed represent an undirected graph.

2.8 Wirfs-Brock et al.'s Work about Contracts

Wirfs-Brock et al. [Wirf 89] proposed a responsibility-driven approach for object-oriented design. It is based on the client/server model which captures the interaction between two entities, the client and server. A client asks the server to provide services. A server provides a set of services corresponding to the request. A contract expresses the ways in which the two entities interact each other. It is constituted of a list of requests that can be made on the server by the client. To fulfil the contract, a client can only make the requests the contract specifies and the server only responds to the corresponding requests. Responsibilities are the individual services provided by the server. The client/server model focuses on what the server does for the client, not how the server does it. The server implementation is encapsulated away from the client. There are three kinds of clients in the model, external clients, subclass clients, and self clients. An external client is an object that sends messages to an instance of the class or the class itself. A subclass client is any class that inherits from the class. Subclass clients help to improve encapsulation by ensuring that all inherited behavior is part of the contract of the subclasses. A self client means any class should be viewed as client of itself.

To summarize, Wirfs-Brock et al.'s notion of contract involves two classes and one or more methods.

2.9 Schrefl and Kappel's Work about Contracts

Cooperation contracts, proposed by Schrefl and Kappel [Schr 91], have a similar syntactic form but a different model than Helm's contract. Cooperation contracts aim to explicitly express the interactions between several objects. They are defined in terms of partner types which participate in a contract and the functionality of the whole contract. Note that in Helm's, Meyer's and Wirfs-Brock et al.'s contracts, only a single object can receive a message. In the Cooperation contracts model, multiple objects can receive a given message. Cooperation contracts involve more than one class and can involve more than one method implementation.

Comparing Cooperation contracts with Helm's contracts, the similarity is that both contracts are defined over multiple types, called partner types and participant types separately. But there exist much differences between them: I) participant types are local to the contract, partner types are external to the contract; ii) the behavior specified in Helm's contract is distributed among the participants, whereas in Cooperation contracts, the behavior is globally attached to the whole; iii) Helm's contract specifies the cooperation relationships of all the participants in the contract, Cooperation contracts reflect the relationships between the partner types and the external clients; and iv) the refinement mechanism in Cooperation contracts only allows a method to be overridden and new methods to be added but does not allow adding new partner types, whereas Helm's contracts supports both of them.

There are also some other researchers that used the same term *contract*, e.g., Budd's interface contracts and subclass contracts [Budd 91]. Helm's Contracts is a generalization of all these contract concepts.

2.10 Hayes and Coleman's Work

For the reason that our final purpose is to formalize Helm's contract to be a specification language of the problem, we discuss hereafter Hayes and Coleman's work [Haye 91] about the formal specification of requirement in an object-oriented context. Their work also referred to the verification of consistency between the dynamic model and functional model of the same problem.

They introduce a set of formal models for object-oriented analysis. The models have mathematical basis so they have a precise semantics and constitute a consistent description of domains. In developing these models, the authors where mainly concerned about the issues of unambiguity, abstraction, and consistency. In other words, models should have a single precise meaning, should not be cluttered with unnecessary details that can influence the design and implementation phases, and it should be possible to verify their consistency with other models of the same system.

These models are improvement of the OMT model by Rumbaugh [Rumb 91] and are illustrated by applying them to the analysis of a simple drawing application which allows the user to draw lines and boxes and move them with some constraints.

OMT uses three different models, namely, the object, dynamic, and functional models. The object model captures structure of the objects. The dynamic model captures the behavior of individual objects. Finally, the functional model represents the behavior of the whole system.

In OMT, the object model uses entity-relationship diagrams to model the object classes, attributes, and relations. The dynamic model is concerned with temporal behavior and uses extended finite state machines. The overall system behavior is shown by the functional

model using data flow diagrams. Entity-relationship diagrams have a formal semantics. State machines and diagrams rely on natural languages and are prone to ambiguity. Moreover, there is no way to check the consistency of the objects or dynamic models with the functional models.

The authors of the current method state that the key to improve the precision of analysis is the introduction of mathematical basis for the above three models. They introduce a new model, the object structure model, that forms a link between the object model and both the functional and dynamic models. The mathematical formalism used is many sorted predicate logic with built in types (like char, natural, and type constructors such as set and sequence).

Every class is represented in the **object structure model** as an abstract structure which precisely states the types of the attributes and relationships in which an object of that class may participate. More precisely, given a class C a record type is associated containing:

- a field "self" of type "C_id", that is the object identifier,
- a field "att" of type "Att_type" for each attribute "att" of type "Att_type" defined in the class, and
- a field "R" of type "set of D_id" for every binary relation from a class C to a class D.

The paper does not show how general relations and inheritance are represented. In the **functional model**, operations on objects are specified in a declarative style by means of pre and postconditions, local to an object and/or global to the system (via parameters of type "System").

The **dynamic model** is based on the Objectcharts notation [Cole 92], an extension of Harel's Statecharts with pre and postconditions associated to events. The effect of events on local attributes is defined using pre and postconditions. Objects communicate together

using synchronous events.

The issue of verification of consistency between models is addressed. Models have formal basis. However, proof of consistency relies on the ingenuity of the analyst. The following approach is suggested. Using the dynamic model, generate event traces of the system behavior. Then, determine if traces implement the declarative specification of the functional model by checking implications of pre and postconditions.

These models are related to the analysis phase of the software development process. The following **analysis procedure** is suggested by the authors:

1) Develop a natural language description of the problem.
2) Build an object model (entity-relationship model) together with a data dictionary of classes, relations, and attributes.
3) Derive an object structure model from the object model.
4) Use the object structure model to define the declarative functional model.
5) Use the object structure model to define a dynamic model together with diagrams showing examples of how events flow through the system in response to system operations.
6) State test event traces and reasoned arguments to show that the dynamic and functional models are consistent.
7) Iterate through the steps looking for missing classes, relationships, attributes, events, etc.

Chapter 3

The Static Semantics of the Contracts Language

3.1 Attribute Grammars

An *attribute grammar* is developed by analyzing the *meaning* of each symbol in a grammar, assigning a meaning to each symbol using the concept of *attribute*, and giving rules of defining the attribute values of the symbols. Each attribute represents a specific (context-sensitive) property of a symbol. The attributes of each symbol can be either synthesized attributes which are passed bottom-up in the derivation tree, or inherited attributes which are passed top-down in the derivation tree (the grammar should be an L-attributed grammar in this case). The idea of defining semantics by associating synthesized attributes with each nonterminal symbol comes from Irons [Iron 61, Iron 63]. Although, synthesized attributes of a symbol (which come from the attributes of the descendants of the symbol) are sufficient to define any function of a derivation tree. The inclusion of inherited attributes of a symbol (which come from the attributes of the ancestors of this symbol) may result in important simplifications.

Let us consider the following context-free grammar which defines a binary number:

(G1)

$$
\begin{aligned}
&B \rightarrow 0\\
&B \rightarrow 1\\
&L \rightarrow B\\
&L \rightarrow LB\\
&N \rightarrow L\\
&N \rightarrow L \bullet L
\end{aligned}
$$

where 0, 1, and "•" are terminal symbols; and B, L, and N are nonterminal symbols representing bits, list of bits, and number respectively. That means any binary number is a sequence of one or more 0's and 1's followed optionally by a radix point and another sequence of one or more 0's and 1's. For example, the string 101.01 is a binary number and it receives the structure pictured in Fig. 2.

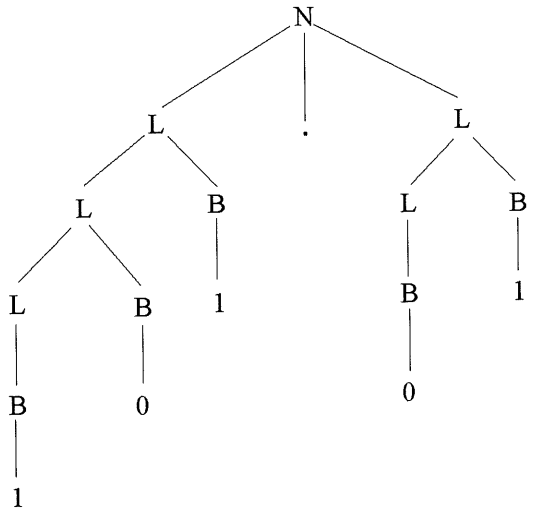

Figure 2: Tree structure of string 101.01

We can give a *meaning* to the above binary notation by assigning *attributes* to the nonterminal symbols as follows: each B has a value v(B) which is an integer, each L has two attributes: length l(L) and value v(L), both are integers, and each number N has a value v(N) which is a rational number. Any desired number of attributes can be given to each nonterminal symbol. And now, we need to give the rules for calculating the value of each attribute according to the semantics of each production. Each rule of grammar G1 can be augmented as follows:

(G2)

$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 1$
$L \rightarrow B$	$v(L) = v(B),\ l(L) = 1$
$L_1 \rightarrow L_2B$	$v(L_1) = 2v(L_2) + v(B),\ l(L_1) = l(L_2) + 1$
$N \rightarrow L$	$v(N) = v(L)$
$N \rightarrow L_1 \bullet L_2$	$v(N) = v(L_1) + v(L_2)/2^{l(L2)}$

Fig. 2 can be expressed by showing the attributes at each level as in Fig. 3.

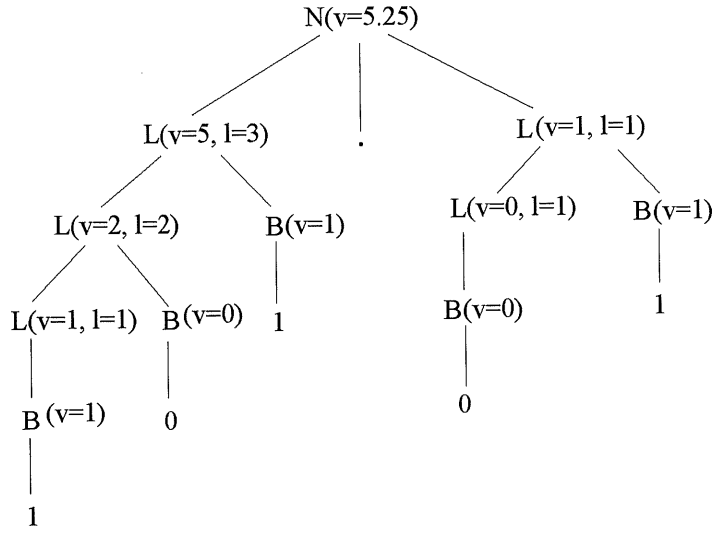

Figure 3: Evaluated tree structure of string 101.01(according to grammar G2)

Up to now, we finished to form the attribute grammar for G1. But let us continue to consider the following thing: how we can define the semantics if we want that the positional characteristics plays a role. For doing this, the following attributes can be defined: each B has two attributes, value v(B) which is a rational number and scale s(B) which is an integer. Each L has three attributes, value v(L) which is a rational number, length l(L), and scale s(L), both are integers. Each N has a value v(N) which is a rational number. Then, the attributes can be defined as follows:

(G3)

$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2^{s(B)}$
$L \rightarrow B$	$v(L) = v(B),\ s(B) = s(L),\ l(L) = 1$
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B),\ s(L_2) = s(L_1)+1,$ $s(B) = s(L_1),\ l(L_1) = l(L_2) +1$
$N \rightarrow L$	$v(N) = v(L),\ s(L) = 0$
$N \rightarrow L_1 \bullet L_2$	$v(N) = v(L_1) + v(L_2),\ s(L_1) = 0,\ s(L_2) = -l(L_2)$

The great differences between the grammar G3 and grammar G2 is that in grammar G2 all attributes were defined when the nonterminal symbol appeared on the left side of the corresponding production, that means there are only synthesized attributes here. All attributes are evaluated in one direction (bottom-up) in the deriving tree. In grammar G3, the attributes can be defined for nonterminal symbols which appear on both sides of the corresponding production. That means that inherited attributes are also included here, so the attributes are evaluated in both direction in the deriving tree. For example, in G3, V(B), V(L), L(L), and V(N) are synthesized attributes and S(B) and S(L) are inherited attributes. The evaluated structure corresponding to the string 101.01 is shown as Fig. 4.

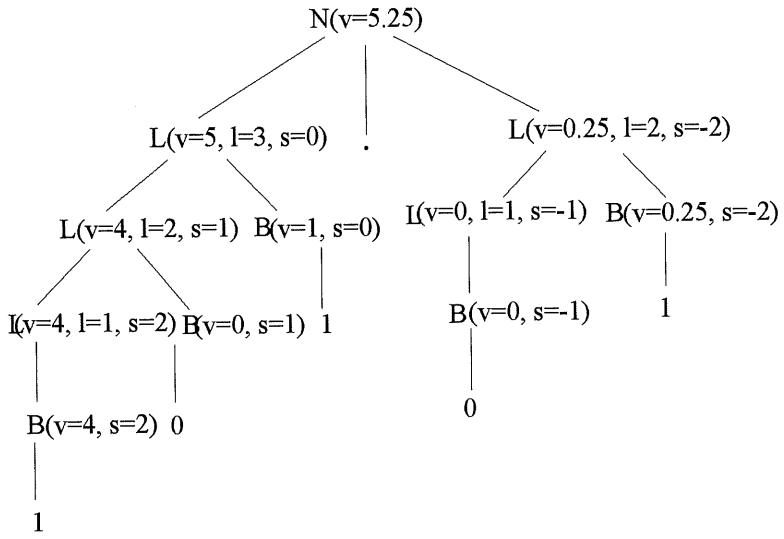

Figure 4: Evaluated tree structure of string 101.01 (according to grammar (G3))

3.2. Formal Definitions of Attribute Grammars

This section gives the formal description of synthesized and inherited attributes using a precise and general setting. Suppose we have a context-free grammar G=(V,N,S,P), where V is a finite vocabulary of terminal and nonterminal symbols, $N \subseteq V$ is a set of nonterminal symbols, $S \in N$ is a start symbol, which appears on the right-hand side of no production rule, and P is a set of production rules. Semantics can be added to G in the following manner: for each terminal or nonterminal symbol $x \in V$, there is an associating finite set A(x) of attributes. Each attribute represents a specific property of symbol x and has a finite range. The attributes of each symbol x can be classified in two parts, the inherited attributes and the synthesized attributes. The evaluation rules of inherited attributes AI(x) are: 1) the start symbol S has no inherited attributes, i.e., $AI(S) = \varnothing$, 2) for a given production, the inherited attributes of the symbol on the right can be evaluated by other symbols in this production, i.e., for the production p: $X_0 \rightarrow X_1 \ldots X_n$, the inherited attributes of X_i ($i <> 0$) are $AI(X_i) = \{ X_i.a \leftarrow f(X_j.b, \ldots, X_k.c)\}$ ($\{j, \ldots, k\} \subseteq \{0, 1, \ldots, n\}-\{i\}$). Here, we use "X.a" to indicate that attribute "a" is an element of AI(X), and 3) for each inherited attribute of the symbol on the left hand, if it exists, it inherits the attributes which have already been evaluated in the previous productions.

The evaluating rules of synthesized attributes AS(x) are: 1) the attributes of a terminal symbol is empty, i.e., $AS(x) = \varnothing$, if x is a terminal symbol, 2) for a given production, the synthesized attributes on the left hand can be evaluated by the attributes of symbols on the right hand or left hand, i.e., for the production p: $X_0 \rightarrow X_1 \ldots X_n$,
$AS(X_0) = \{X_0.a \leftarrow f(X_j.b, \ldots, X_k.c)\}$ ($\{j, \ldots, k\} \subseteq \{1, \ldots, n\}$).

For any given production p, there may be a condition $B(X_i.a,\ldots,X_j.b)$, in addition to the attribution rules. A condition $B(X_i.a,\ldots,X_j.b)$ specifies a contextual property according to the static semantics. It must be fulfiled by a sentence.

Definition 3.1. An attribute grammar is a 4-tuple, AG = (G,A,R,B), where G = (V,N,S,P) is a context-free grammar, $A = \cup A(X)$ ($X \in T \cup N$) is a finite set of attributes, $R = \cup R(p)$ ($p \in P$) is a finite set of attribution rules and $B = \cup B(p)$ ($p \in P$) is a finite set of conditions.

Definition 3.2. For each p: $X_0 \rightarrow X_1 \ldots X_n \in P$, the set of defining occurrences of attributes is $AF(p) = \{ X_i.a \mid i \in \{0, 1,\ldots, n\}$ and $X_i.a \leftarrow f(\ldots) \in R(p) \}$. An attribute $X_i.a$ (i=0, 1,...,n) is called derived or synthesized if there exists a production p: $X_i \rightarrow \chi$ and $X_i.a$ is in AF(p); it is called inherited if there exists a production q: $Y \rightarrow \mu X_i \upsilon$ and $X_i.a$ is in AF(q).

Synthesized attributes of a symbol are evaluated by considering the subtree derived from this symbol in the structure tree. Inherited attributes are evaluated by considering the environment of this symbol.

3.3. An Attribute Grammar for Contracts

In the following, we adopt the conventions, functions, and notations from Waite and Goos [Wait 85] to give the attribute grammar for the Contracts language. The basis document for the syntax of this grammar is Holland's paper [Holl 92]. Each production $p \in P$ is expressed using EBNF notation and marked by the keyword **rule**. The attributes of each production p are marked by the keyword **attribution** and conditions may follow the keyword **condition**.

3.3.1. Utilities

For simplification purposes, the following conventions are used. A synthesized attribute of the left-hand side of a production may be omitted if there is only one symbol on the right-hand side and this attribute is same-named with that of right hand, e.g., for the rule statements ::= statement, we omit the attribution rules. Simple assignments of inherited attributes of the left-hand side to same-named inherited attributes of any number of right-hand symbols may be omitted.

We also use the assumption that for every record type R used to describe attributes, there exists a function N_R which has the parameters corresponding to the fields of record R. This function creates a new record of type R and assigns the parameter values to the fields of R.

For representing every definition of an identifier, the following variant record is used:

```
type
definition_class = (
                object_definition,
                type_definition,
                unknown_definition);
definition = record
        uid:integer;
        ident:symbol;
        case k:definition_class of
            object_definition:(object_type:mode);
            type_definition:(defined_type:mode);
            unknown_definition:()
end;
```

The variant *object_definition* is used to refer to object identifiers. Objects are the concrete instances of values that operations are operated upon. The variant *type_definition* is used to refer to type identifiers. The definition class *unknown_definition* is used to deliver a value if no definition is available for an identifier.

Records of type *definition* are collected into linear lists, referenced as the *environment* and *definition* attributes, by every construct that uses an identifier. The type of the environment attribute is:

```
definition_table = ↑dt_element;
dt_element = record
    first:definition;
    rest:definition_table
end;
```

Types classify values according to the operations which can be performed on them and the applicable coercions. The following record is used to build attributes describing types:

```
type
type_class = (
    bad_type, void_type, bool_type, int_type, real_type,
    ref_type, set_type, proc_type, obligation_type,
    unidentified_type, identified_type, contract_type);
mode = record
case k:type_class of
    bad_type, void_type, bool_type, int_type, real_type:();
    ref_type:(target:↑mode);
    set_type:(element:↑mode );
    proc_type:(parameters:definition_table; result: ↑mode);
```

```
    contract_type:(parameters:definition_table);
    obligation_type:(
            variables:definition_table,
            methods:definition_table);
    unidentified_type:(identifier:symbol);
    identified_type:(definition:integer)
end;
```

Type *bad_type* is used to indicate that errors have made it impossible to determine the proper type. Type *void_type* is used in the case that a result of a procedure is to be discarded. The type *unidentified_type* is used when a type identifier occuring in a declaration is not yet identified. The type *identified_type* is used when a type identifier occuring in definition has already been identified.

In the following, we introduce functions used in this thesis. They come from [Wait 85].

If we want to look for the definition of a symbol in a definition table, the function *current_definition* can be used. It searches the environment sequentially from left to right and selects the first definition for the desired identifier.

```
function current_definition(s:symbol; dt:definition_table):definition;
begin
  if dt=nil
  then current_definition:=nil
  else if dt↑.first.ident=s
       then current_definition:=dt↑.first
       else current_definition:=current_definition(s, dt↑.rest)
end;
```

If we want to find the type defined by an identifier in a definition table, the function *identify_type* can be used. It searches the environment sequentially from left to right and selects the first type defined by the identifier.

```
function identify_type(s:symbol; dt:definition_table):mode;
(* Find the type defined by an identifier *)
begin
  if dt=nil then identify_type:=N_mode(bad_type)
  else with dt↑.first do
    if s<>ident then
        identify_type:=identify_type(dt↑.rest)
    else (* s=ident *)
    if k<>type_definition then
        identify_type:=N_mode(bad_type)
    else identify_type:=N_mode(identified_type, uid)
end;
```

The function *lookup* finds a definition from its unique integer identifier.

```
function lookup(def: integer; dt: definition_table): definition;
begin
  if dt=nil
  then "attribute grammar error"
  else
    if dt↑.first.uid=def
    then lookup := dt↑.first
    else lookup := lookup(def, dt↑.rest)
end;
```

In order to check the consistency of expressions and statements (e.g., expression::=name) and identify the operator used in expressions (e.g., expression::=expression+expression), the types of the operands must be taken into account. Two attributes *primode* and *postmode* are defined for this purpose. The attribute primode describes the type determined directly from a node and its descendants. It is a synthesized attribute. The attribute postmode describes the type expected when a result is used as an operand by other nodes. It is a type required in a context and this attribute is inherited. The function *deproc* is used to obtain the primode of a name.

Function *deproc(t)* recursively tests the *type_class* of *t*. If *t* belongs to a *non-proc_type* or *proc_type* with parameters, the function returns *t* itself, otherwise, if *t* belongs to a *proc_type* without parameters, the function keeps testing the *type_class* of t.result↑.

```
function deproc(t:mode):mode;
begin
  if t.k <> proc_type then deproc := t
  else if t.parameters <> nil then deproc := t
  else deproc := deproc(t.result↑)
end;
```

Function *compare_parameters(f1, f2)* tests whether or not the two lists *f1*, *f2* include the same numbers of elements. If not the function returns false, otherwise, it further tests whether or not each pair of the elements which have the same sequence number in *f1*, *f2* seperately are type equivalent. If yes, the function returns true, otherwise it returns false.

```
function compare_parameters(f1, f2: definition_table):boolean;
(* Compare parameter lists for equivalent types *)
begin
  if f1=nil then compare_parameters:=f2=nil
  else if f2=nil then compare_parameters:=false
  else
    compare_parameters:=
        type_equivalent(
            f1↑.first.object_type,
            f2↑.first.object_type) and
    compare_parameters(f1↑.rest, f2↑.rest)
end;
```

Function *type_equivalent* tests whether or not *t1* and *t2* are semantically equivalent. If *t1* and *t2* do not belong to the same type class, the function returns false, otherwise the function returns true except in the following three cases which need further test: 1) if they are both *ref_type*, the function returns true if their target types are *type_equivalent*, otherwise, the function returns false, 2) if they are both *proc_type*, the function returns true if their parameters and their result are *type_equivalent*, and 3) if they are both *identified_type*, the function returns true if the definitions of these two types are equal.

```
function type_equivalent(t1, t2:mode):boolean:
(* Compare two types for equivalence *)
begin
  if t1.k<>t2.k then type_equivalent:=false
  else /* t1.k=t2.k */
  case t1.k of
  ref_type: type_equivalent:=
```

```
    type_equivalent(t1.target↑, t2.target↑)
  proc_type: type_equivalent:=
    compare_parameters(t1.parameters, t2.parameters) and
    type_equivalent(t1.result↑, t2.result↑);
  identified_type:
    type_equivalent:=t1.definition=t2.definition
  otherwise type_equivalent:=true
  end
end;
```

Coercible is one kind of relation expressed by recursive functions over types: a type *t1* is coercible to a type *t2* if it is either compatible with *t2* or can be converted to *t2* by a sequence of coercions. Function *coercible(t1, t2)* means *t1* is coercible to *t2* in the following cases: 1) *t1* and *t2* are semantically equivalent or 2) if they are not semantically equivalent then i) *t1* can be any type if the type class of *t2* is *void_type* or *bad_type*, ii) *t1* belongs to *bad_type*, iii) *t2* can be *real_type* if *t1* is *int_type*, iv) *t1.target*↑ must be coercible to *t2* if *t1* is *ref_type* or v) *t1.result*↑ must be coercible to *t2* if *t1* is *proc_type* with no parameters.

```
function coercible(t1, t2 : mode):boolean;
begin
  if type_equivalent(t1, t2) or
    t2.k=void_type or t2.k=bad_type
  then coercible: =true
  else
  case t1.k of
  bad_type:coercible:=true
```

```
  int_type:coercible:=t2.k=real_type
  ref_type:coercible:=coercible(t1.target↑, t2)
  proc_type:coercible : =
    t1.parameters=nil and coercible(t1.result↑, t2)
  otherwise coercible:=false
  end;
end;
```

Function *deref(t)* removes all levels of reference from type *t*. It recursively checks the type class of *t*. If *t* belongs to a *non-ref_type*, the function returns *t* itself. Otherwise, if *t* belongs to a *ref_type*, the function keeps testing the type class of *t.target*↑.

```
function deref (t: mode): mode;
(* Remove all levels of reference to a type*)
begin (*deref*)
if t.k <> ref_type then deref := t
else deref := deref (t.target↑);
end; ( * deref * )
```

Function *base_type(t)* tests the type of *t*. It returns *t* if *t* does not belong to *ref_type* or *proc_type* without parameters. Otherwise, it recursively tests the type of the target or the result of t.

```
function base_type ( t : mode) : mode;
( * Remove all levels of reference and procedure call from a type * )
begin ( * basetype * )
if t.k = ref_type then base_type := base_type (t.target↑)
else if t.k = proc_type then
  if t.parameters <> nil then base_type := t
  else base_type :=base_type (t.result↑)
else base_type := t
end; ( * base_type * )
```

Function *balance(t1, t2)* gets the balance type of *t1* and *t2*. It works as follows: i) it returns *t2* if *t1* is coercible to *t2*, ii) it returns *t1* if *t2* is coercible to *t1*, iii) if *t1* is coercible to the base type of *t2*, the result type is a dereferencing (remove all references) and/or deproceduring (remove all levels of procedure) of *t2*, and if *t2* is coercible to the base type of *t1*, the same operation will be done with *t1*.

```
function balance ( t1, t2 : mode ):mode;
( * Obtain the representative a priori type of t1, t2 * )
begin ( * balance * )
if coercible ( t1, t2 ) then balance := t2
else if coercible ( t2, t1) then balance := t1
else if coercible (t1, base_type ( t2)) then
   case t2.k of
     ref_type : balance := balance (t1, t2.target↑);
     proc_type := balance (t1, t2.result↑)
     end
else if coercible ( t2, base_type (t1)) then
```

```
  case t1.k of
    ref_type : balance := balance ( t1.target↑, t2);
    proc_type := balance (t1.result↑, t2)
    end
else N_mode ( void_type);
end; ( * balance * )
```

3.3.2. Productions

3.3.2.1. Basic Symbols

rule letter ::= 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|
'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|
'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|
'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'.

A letter is one of the twenty-six letters, lower or upper case.

rule digit ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.

A digit is one of the ten numbers of zero to nine.

rule identifier :: = letter (letter | digit | '-')*.

An identifier is a string of letters, digits or '-' beginning with a letter.

rule integer ::= digit.

rule integer ::= digit integer.

An integer is defined by a digit or recursively by a digit followed by another integer.

rule real ::= integer '.' integer.

A real number is defined by two integers with a dot separating them.

3.3.2.2. Contracts Structure

rule contracts ::= contracts ';' contract.

attribution

contracts[1].definitions ← contracts[2].definitions & contract.definitions.

Several contracts can be defined in a single file.

rule contract ::=.

rule contract ::=

'contract' identifier

'participants' participant_declarations

inclusion_part

refinement_part

contract_body

invariants

instantiation

'end' 'contract'

attribution

contract.definitions ←

N_definition (

gennum,

identifier.symbol,

```
        object_definition,
        N_mode (
            contract_type,
            participant_declarations.definition));
    contract_body.environment ←
        complete_env (participant_declarations.definitions, contract_body.definitions)&
        contract_body.definitions & inclusion_part.definitions &
        refinement_part.definitions;
condition
    unambiguous(participant_declarations.definitions & inclusion_part.definitions &
                    refinement_part.definitions & contract_body.definitions);
```

A contract is defined in terms of a unique name (contract_name), declarations of named and typed participants (participant_declarations), a list of obligations for every participant type (contract_body), global invariants (invariants), and a section describing how the contract is instantiated (instantiation). A contract can also be defined by including (inclusion_part) or refining (refinement_part) some already existing contracts. *Gennum* is a function which used to generate a unique integer. Each invocation of *gennum* yields a new integer.

The function *complete_env* is recursive and traverses the definitions in *participant_declarations.definitions* seeking participant declarations with unidentified types. Whenever one is found, the current definition of the type identifier is obtained from *contract_body.definitions* by using the function *identify_type*.

The function unambiguous is used to verify that every identifier is defined once in *participant_declarations.definitions*, *inclusion_part.definitions*, *refinement_part.definitions*, and *contract_body.definitions*. l_1 & l_2 to expresses the concatenation of two lists l_1 and l_2.

rule inclusion_part ::= .

attribution

inclusion_part.definitions ← nil.

A contract can be defined without including other contracts.

rule inclusion_part ::= inclusions ';'.

rule inclusions ::= inclusion.

rule inclusions ::= inclusions ';' inclusion.

attribution

inclusions[1].definitions ← inclusions[2].definitions & inclusion.definitions.

A contract can be defined by including one or more contracts.

rule inclusion::=

'includes' identifier.

attribution

inclusion.corres_def ←

current_definition(

identifier.symbol,

inclusion.environment);

condition

inclusion.corres_def.object_type.k=contract_type;

rule inclusion::=

'includes' identifier '(' correspondances ')'.

attribution

inclusion.corres_def ←

current_definition(

identifier.symbol,

inclusion.environment);

condition

inclusion.corres_def.object_type.k=contract_type;

compare_parameters(

correspondances.definitions,

inclusion.corres_def.object_type.parameters);

An inclusion clause gives the name of an included contract in a new contract and (if there is) the correspondance between the participants of the new contract and those of the included one.

rule correspondances ::= expressions.

Correspondances describe the correspondance relation between the participants that belong to the new contract and those of the included contract.

rule refinement_part ::= .

A contract can be defined without refining other contracts.

rule refinement_part ::= refinements ';'.

rule refinements ::= refinement.

rule refinements ::= refinements ';' refinement.

attribution

refinements[1].definitions ← refinements[2].definitions & refinement.definitions.

A contract can be defined by refining one or more contracts.

rule refinement ::= 'refines' identifier.

attribution

refinement.corres_def ←

```
        current_definition(
            identifier.symbol,
            refinement.environment);
condition
  refinement.corres_def.object_type.k=contract_type;

rule refinement ::= 'refines' identifier '(' correspondances ')'.
attribution
   refinement.corres_def ←
        current_definition(
            identifier.symbol,
            refinement.environment);
condition
  refinement.corres_def.object_type.k=contract_type;
  compare_parameters(
    correspondances.definitions,
    refinement.corres_def.object_type.parameters);
```

A refinement gives the name of a refined contract in a new contract and (if there is) the correspondance between the participants of new contract and those of the refined one.

```
rule instantiation ::= 'instantiation' statements.
```

3.3.2.2.1. Participant Declarations

```
rule participant_declarations ::= participant_declaration.
```

rule participant_declarations ::=

participant_declarations ';' participant_declaration.

attribution

/* collect all the participant declarations */

participant_declarations[1].definitions ←

participant_declarations[2].definitions &

participant_declaration.definitions;

Participant_declarations declare a list of participating objects in the contract. Each participant declaration is separated by the symbol ";". Participant_declaration is used to create participants in the contract.

A participant declaration creates participating object(s) of the specified participant type.

rule participant_declaration::=

identifier ':' participant_type_specification.

attribution

participant_declaration.definitions ←

N_definition(

gennum,

identifier.symbol,

object_definition,

participant_type_specification.representation);

A participant declaration explicitly names a participant (the symbol identifier is a variable used to be the name of this participant) and the obligations (participant_type_specification) of this participant.

The symbol identifieris is a variable that refers to a participant of type *participant_type_specification*.

Type identifiers occuring in participant type declarations are given the *unidentified_type* variant.

rule participant_type_specification ::= identifier.

attribution

participant_type_specification.representation ←

N_mode(unidentified_type, identifier.symbol);

rule participant_type_specification ::='set' '(' identifier ')'.

attribution

participant_type_specification.representation ←

N_mode(

set_type,

N_mode(unidentified_type, identifier.symbol));

A participant type specification can be an anonymous set of participants (of type identifier) using the keyword 'set'.

3.3.2.2.2. Contract Body

rule contract_body ::= obligation_specifications.

attribution

contract_body.definitions ←

obligation_specifications.definitions;

Contract_body includes the obligation definitions of all participant objects in a contract (obligation_specifications).

rule obligation_specifications ::= obligation_specification.

rule obligation_specifications ::= obligation_specifications ';' obligation_specification.

attribution

obligation_specifications[1].definitions ←

obligation_specifications[2].definitions &

obligation_specification.definitions;

Obligation_specifications can be used to give the obligation definition of one participant (obligation_specification) or a list of participants (obligation_specifications ';' obligation_specification).

rule obligation_specification ::= identifier 'supports' '[' obligations ']'.

attribution

obligation_specification.definitions ←

N_definition(

gennum,

identifier.symbol,

type_definition,

obligations.representation);

Obligation_specification contains the definition of obligations for each participant (identifier) declared in participant_declarations.

rule obligations ::= variable_declarations ';' methods.

attribution

obligations.representation ←

N_mode(

obligation_type,

variable_declarations.definitions,

methods.definitions);

methods.environment ←

complete_env(

variable_declarations.definitions,

variable_declarations.definitions &

methods.definitions & obligations.environment) &

methods.definitions &

obligations.environment;

condition

unambiguous(variable_declarations.definitions);

unambiguous(methods.definitions);

Obligations describes how each participant contributes to the contract. It includes type obligations in which a list of instance variables of certain types are defined (variable_declarations) and causal obligations (methods) which make explicit behavioral dependencies between the objects in a contract.

rule obligations ::= methods.

Participants in a contract can provide methods without supporting instance variables.

rule variable_declarations ::= variable_declaration.

rule variable_declarations ::= variable_declarations ';' variable_declaration.

attribution

variable_declarations[1].definitions ←

variable_declarations[2].definitions &

variable_declaration.definitions;

Variable_declarations declare one instance variable (variable_declaration) or several instance variables (variable_declarations ';' variable_declaration). Each variable is separated by the symbol ';'.

A variable declaration creates a reference to a given type (i.e., a variable referring to an undefined value.

rule variable_declaration ::= identifier ':' type_specification.

attribution

```
variable_declaration.definitions ←
  N_definition(
      gennum,
      identifier.symbol,
      object_definition,
      N_mode(
          ref_type,
          type_specification.representation));
```

Variable_declaration declares a variable by defining its name (identifier) and its type (type_specification).

rule type_specification ::= participant_type_specification.

Type_specification can be of the type participant_type_specification, refered before.

rule type_specification ::= 'integer'.

attribution

```
type_specification.representation ← N_mode(int_type);
```

rule type_specification ::= 'real'.

attribution

```
type_specification.representation ← N_mode(real_type);
```

rule type_specification ::= 'bool'.

attribution

type_specification.representation ← N_mode(bool_type);

Type_specification can be one of the standard type such as *integer*, *real*, or *boolean*.

rule methods ::= method.

rule methods ::= methods ';' method.

attribution

methods[1].definitions ←

methods[2].definitions & method.definitions;

Methods can include one (defined by method) or several methods (recursively defined by methods ';' method) definitions.

rule method ::=

identifier

'(' parameter_type_list ')'

result_type

statement_list.

attribution

method.definitions ←

N_definition(

gennum,

identifier.symbol,

object_definition,

N_mode(

proc_type,

parameter_type_list.definitions,

result_type.representation));

statement_list.environment ←
method.environment;
statement_list.postmode ← result_type.representation;

A method definition includes four parts: a unique name (identifier), a list of parameters (parameter_type_list), a type of the method (result_type), and a list of statements (statement_list).

rule parameter_type_list ::= .
attribution
parameter_type_list.definitions ← nil;

rule parameter_type_list ::= parameter_declarations.

There may be no parameter in the method, so a parameter_type_list can be nil, or there can be one or several parameters in the method. These parameters are declared in the parameter_type_list (parameter_declarations).

rule parameter_declarations ::= parameter_declaration.

rule parameter_declarations ::= parameter_declarations ';' parameter_declaration.

Parameter_declarations declare one (parameter_declaration) or more than one parameter (recursively defined by parameter_declarations ';' parameter_declaration).

rule parameter_declaration ::= identifier ':' type_specification.
attribution
parameter_declaration.definitions ←
N_definition(
gennum,
identifier.symbol,

object_definition,

N_mode(

ref_type,

type_specification.representation));

Parameter_declaration declares a parameter variable by giving its name (identifier) and its type (type_specification).

rule result_type ::= .

attribution

result_type.representation ← nil;

rule result_type ::= ':' 'void'.

attribution

result_type.representation ← N_mode(void_type);

rule result_type ::= ':' type_specification.

attribution

result_type.representation ← type_specification.representation;

The result returned after executing the method can be a type of *void* or of a type given by *type_specification*.

rule statement_list :: = .

rule statement_list ::= '[' statements']'.

condition

coercible(

statements.primode,

statement_list.postmode);

Statement_list can include no statement (.) or some (one or more) statements (statements).

rule statements ::= statement.

rule statements ::= statements ';' statement.

attribution

statements[1].primode ← statement.primode;

Statements can include only one statement (statement) or several statements (statements).

rule statement ::= .

rule statement ::= name ':=' expression.

attribution

name.postmode ← name.primode;

expression.postmode ←

if name.postmode.k<>ref_type

then N_mode(bad_type)

else name.postmode.target↑;

condition

name.postmode.k=ref_type;

A statement can be an assignment. After the execution of an assignment statement, the value associated to the name on its left-hand side is replaced by the result of the evaluation of the expression on the right-hand side.

rule statement ::= condition.

rule condition ::= '{' expression '}'.

attribution

expression.postmode ← N_mode (bool_type);

condition.primode ← N_mode(void_type);

A statement can be an expression corresponding to a condition to make true.

rule statement ::= name '→' method_call.

attribution

method_call.environment ←

lookup(name.primode.target↑.definition,

statement.environment).defined_type.methods

& statement.environment;

method_call.postmode ← N_mode(void_type);

rule statement ::= method_call.

attribution

method_call.postmode ← N_mode(void_type);

rule statement ::= 'if' expression 'then' '{' statements '}'.

attribution

statement.primode ← N_mode(void_type);

expression.postmode ← N_mode(bool_type);

A statement can be an if statement using the keywords "if" and "then". The execution of this statement is: the test condition (expression) is evaluated first. If it is true, the statements are executed; otherwise, these statements are skipped and the statements following are executed.

rule statement ::= 'for all' identifier 'in' type_specification '{' statements '}'.

attribution

statements.environment ←

complete_env(

N_definition(
gennum,
identifier.symbol,
object_definition,
N_mode(ref_type,
type_specification.representation)) & statement.environment,
statement.environment).

A statement can be a for statement using the keywords "for" and "in". The execution of this statement is: for each variable refered by an identifier which belongs to the set refered by type_specification, execute all statements in the rule.

rule statement ::= 'while' expression 'do' '[' statements ']' ;

attribution

statement.primode ← N_mode (void_type);

expression.postmode ← N_mode (bool_type);

A statement can be an iteration statement using the keyword "while". The execution of this statement is: the expression is evaluated and if the result is true the statements are performed. When the expression becomes false, the program proceeds directly to the following statements.

rule statement ::= 'return' expression.

attribution

statement.primode ← expression.primode;

A statement can be a return statement using the keyword "return". The execution of this statement is: evaluate the value of the expression and return this value and it can be used in a certain scope.

3.3.2.2.3. Expressions

rule name ::= identifier.

attribution

name.primode ←

if current_definition(

identifier.symbol,

name.environment).k<>object_definition

then

N_mode(bad_type)

else current_definition(

identifier.symbol,

name.environment).object_type;

condition

current_definition(

identifier.symbol,

name.environment).k=object_definition

The name of any object is given by an identifier.

rule expression ::= name.

attribution

expression.primode ← name.primode;

condition

coercible(expression.primode, expression.postmode);

An expression can just be a name.

rule expression ::= expression op expression.

attribution

expression[2].environment ← expression[1].environment;

expression[3].environment ← expression[1].environment;

expression[1].primode ←

if coercible(expression[2].primode, int_type) **and**

coercible(expression[3].primode, int_type)

then N_mode(int_type)

else N_mode(real_type);

expression[2].postmode ← expression[1].primode;

expression[3].postmode ← expression[1].primode;

condition

coercible(expression[1].primode, expression[1].postmode);

rule op ::= '+'| '-'|'*'|'/'.

An expression can be two expressions connected by one of the operators '+', '-', '*', or '/'.

rule expression ::= 'true' | 'false'.

attribution

expression.primode ← N_mode(bool_type);

condition

coercible(expression.primode, expression.postmode);

rule expression::= integer.

attribution

expression.primode ← N_mode(int_type);

condition

coercible(expression.primode, expression.postmode);

rule expression::= real.

attribution

expression.primode ← N_mode(real_type);

condition

coercible(expression.primode, expression.postmode);

An expression can be simply a boolean, an integer or a real value.

rule expression ::= expression ['and' |'or'] expression.

attribution

expression[1].primode ←

if coercible(expression[2].primode, bool_type) **and**

coercible(expression[3].primode, bool_type)

then N_mode(bool_type)

else N_mode(bad_type);

condition

coercible(expression[1].primode, expression[1].postmode);

An expression (expression[1]) can be a logic expression in which two logic expressions (expression[2] and expression[3]) are connected by the keywords *and* or *or*. The operator *and* yields a *true* value for the expression on its left hand if both expressions on its right-hand yield *true*. If either or both expressions on the right evaluate to *false*, the value of the expression on the left is *false*. The *or* operator evaluates to *true* for the expression on the left if either or both expressions on the right are *true*; the value of the expression on the left is *false* only if both expressions on the right evaluate to *false*.

rule expression ::= 'not' '(' expression ')'.

attribution

expression[1].primode ←

if coercible(expression[2].primode, bool_type)

then N_mode(bool_type)

else N_mode(bad_type);

condition

coercible(expression[1].primode, expression[1].postmode);

An expression (expression[1]) can be a logic expression using the keyword *not* in front of an expression (expression[2]). The value of the expression on the left is *true* when the value of the expression on the right is *false*, otherwise, it has the value *false*.

rule expression ::= expression ['=' | '<>'] expression.

attribution

expression[2].postmode ←

expression[3].postmode ←

deref(

balance(

expression[2].primode,

expression[3].primode));

expression[1].primode ← N_mode(bool_type);

condition

coercible(expression[1].primode, expression[1].postmode);

An expression can be a relational expression in which two expressions are connected by the relational operator equality ('=') or not equality ('<>'). The '=' yields the value *true* if and only if the values of two expressions on the right are equal. The '<>' yields the value *true* if and only if the values of two expressions on the right are not equal.

rule expression ::= expression ['=>'| '<=>'] expression.

attribution

expression[1].primode ←

if coercible(expression[2].primode, bool_type) **and**

coercible(expression[3].primode, bool_type)

then N_mode(bool_type)

else N_mode(bad_type);

condition

coercible(expression[1].primode, expression[1].postmode);

An expression can be two expressions connected by the symbol "=>" or " <=> ", where "=>" means implication and " <=> " means equivalence.

rule expression ::= expression '⊆ ' expression.

attribution

expression[1].primode ← N_mode(bool_type);

expression[2].postmode ← expression[3].primode;

expression[3].postmode ← N_mode(void_type);

condition

coercible(expression[1].primode, expression[1].postmode);

An expression can be two expressions connected by the symbol " ⊆ " to express that a set which has the properties described by the former expression (expression[2]) is a subset of the set which has the properties described by the later one (expression[3]).

rule expression ::=

'∀' identifier '∈ ' type_specification ':' '[' statements ']'.

attribution

statements.environment ←

N_definition(

gennum,

identifier.symbol,

object_definition,

N_mode(

ref_type,

type_specification.representation)) &

expression.environment;

condition

coercible(expression.primode, expression.postmode);

This expression declares an object that belongs to a set of a certain type (type_specification). This object, named by the identifier, has the properties described by the expression following the symbol ":".

rule expression ::=

'∃' identifier '∈ ' type_specification ':' '[' statements. ']'

attribution

statements.environment ←

N_definition(

gennum,

identifier.symbol,

object_definition,

N_mode(

ref_type,

type_specification.representation)) &

expression.environment;

condition

coercible(expression.primode, expression.postmode);

This expression declares there exists at least one object that belongs to the set of a certain type (type_specification). This object, named by the identifier, has the properties described by the expression following the symbol ":".

rule expression ::= name '∈ ' name.

attribution

expression.primode ← N_mode(bool_type);

name[2].primode.k =set_type;

name[1].primode = name[2].primode.element↑;

condition

coercible(expression.primode, expression.postmode);

This expression means that the object named by the former identifier belongs to the set named by the later identifier.

rule expression ::= name '→' method_call.

attribution

method_call.environment ←

lookup(name.primode.target↑.definition, expression.environment).defined_type.methods & expression.environment;

method_call.postmode ← expression.postmode;

condition

name.primode.k=ref_type;

name.primode.target↑=identified_type;

lookup(name.primode.target↑.definition, expression.environment).k=type_definition;

lookup(name.primode.target↑.definition, expression.environment).defined_type.k = obligation_type;

An expression can be a method call. This expression results in a participant (refered by name) to invoke a method which has the name given in method_call. In the above, the symbol *name* synthesizes an attribute *primode* representing the type of the identifier denoted by name (see the rule for *name* at the beginning of this section). The fact that the class of the returned definition is *type_definition* is insured by the third condition (⋯.k=type_definition). The fact that the class of the returned type is *obligation_type* is

insured by the fourth condition (…k=obligation_type). Note that a type specification of the class *obligation_type* has an attribute *method* which concatenating with the attribute *expression.environment* constitutes the attribute *environment* inherited by the symbol *method_call* in the above.

rule expression ::= method_call.

attribution

method_call.environment ← expression.environment;

method_call.postmode ← expression.postmode;

An expression can be a method_call. This expression results in the invocation of the corresponding method which has the name given in method_call.

rule method_call ::= name '(' argument_list ')'.

condition

name.primode.k=proc_type;

compare_parameters(argument_list.definitions, name.primode.parameters);

coercible(name.primode.result↑, method_call.postmode);

A method call must indicate the name (name) of the called method, the correspondance between the arguments (argument_list, there can be no) and the parameters (parameter_type_list) listed in the definition of the method, the correspondance between the arguments and the parameters is made according to their appearance sequence. The caller method and the callee method should agree on the number and types of the parameters. The names may be the same or different.

rule argument_list ::= .

attribution

argument_list.definitions ← nil;

rule argument_list ::= arguments.

rule arguments ::= argument.

rule arguments ::= arguments ';' argument.

attribution

argument[1].definitions ←

arguments[2].definitions & argument.definitions;

Corresponding to the parameter_declarations in the definition of the method, argument_list can include no argument (.), one argument (argument) or several arguments (recursively defined by arguments ';' argument.

rule argument ::= expression.

attribution

expression.postmode ← N_mode(void_type);

An argument is an expression.

3.3.2.2.4 Invariants

rule invariants ::= .

rule invariants ::= 'invariant' invariant_list.

rule invariant_list ::= inv_statement.

rule invariant_list ::= invariant_list ';' inv_statement.

rule inv_statement ::= expression.

attribution

expression.postmode ← N_mode(bool_type);

Chapter 4

Using Lex and Yacc to Contracts

For analysing programs with structured input, there are two tasks that need to be done: 1) divide the input into meaningful units which are often called tokens, this task is known as lexical analysis, and 2) discover the relationships among the units, namely, finding the expressions, statements, declarations, and blocks. This task is known as parsing. The relationships are defined by a list of grammar rules. Lex and Yacc [Levi 92] are two tools to accomplish these two tasks.

Lex is a tool for building lexical analyzer or lexers. It generates a C routine which can identify different tokens by taking a set of descriptions. The C routine is the lexer and the set of descriptions is called a Lex specification. A Lex specification is a set of regular expressions which the lexer matches with the input. Each time one of the regular expressions is matched, the lex program invokes C code which does something for the matched text. This C code is written by the user of Lex. In this way, a lexer divides the input into tokens. These output tokens can either be the "end product" or be processed further, often by Yacc.

Yacc takes as input a series of rules which is called a grammar and generates a parser that recognizes valid "sentences" in the grammar. Two main actions of Yacc are shift and reduction. Shift action occurs when Yacc reads a token that doesn't complete a rule. Reduction action occurs when Yacc finds all the symbols that constitute the right-hand side of a rule. Yacc can not deal with ambiguous grammars and grammars that need more than one token of lookahead to tell whether it has matched a rule. When there are conflicts in a Yacc grammar, Yacc reports shift/reduce and reduce/reduce errors by pointing them in an output file.

Both Yacc grammar and Lex specification has the same three-part structure which includes: the description, rules, and user subroutines sections.

Our work includes the implementation of the lexical and syntax analysis of Contracts using Lex and Yacc. The following sections describe the Lex regular expressions and Yacc grammar for Contracts language.

4.1 Lex Specification of Contracts

Our Lex specification of Contracts consists of three parts: the definition, rules and user subroutines sections. It has the form of:

...definition section...
%%
...rules section...
%%
...user subroutines...

The definition section for Contracts is bracketed by "%{" and "%}", which means that the bracketed contents is copied into the lexer directly. The contents bracketed here is #include "y.tab.h". The file y.tab.h is generated by Yacc and contains all of the token definitions. It must be included here because Lex and Yacc must agree with what the token codes are.

The rules section of the Contracts language identifies the different tokens that Contracts uses. This includes a fixed set of reserved words and some other connectional tokens such as identifiers, numbers, and operators.

The last section includes error and main routines. The function yyerror() reports where the error is, including the current line number, current token, and an error message. The main

routine opens a file named on the command line and calls the parser. The returned value reports whether the parse succeeded or failed.

Our Lex specification of Contracts is included in Appendix A.

4.2 Yacc Grammar of Contracts

Our Yacc grammar of Contracts uses the same structure as the Lex specification. The first two sections are required, but can be empty. The third section may be omitted and it is omitted here.

The definition section of the Contracts language contains declarations %token and %left. %token declares terminal symbols that the lexer passes to the parser. All symbols used as tokens must be defined explicitly in the definition section. Tokens can also be declared by %left declarations. The %left is used to declare an operator which is left associative.

The rule section contains all grammar rules of the Contracts language such as the rules of declaration of each participant, inclusion, refinement of the existed contracts, obligations of each participant, invariants and instantiation of a contract.

The Yacc grammar is included in Appendix B.

4.3 Parser-Lexer Communication

When a Lex scanner and a Yacc parser are used together, the parser is the high-level routine. The parser calls the lexer whenever it needs a token from the input. The lexer and parser must agree on what the token codes are. This is solved by letting Yacc define the

token codes in file y.tab.h and the lexer use these token number definitions by including the file y.tab.h.

4.4 Running Lex and Yacc

The Lex specification file is called contracts.l and the yacc grammar file is called contracts.y. For building the output, one need to do following in UNIX:

```
% yacc -d contracts.y          # makes y.tab.c and "y.tab.h"
% lex contracts.l              # makes lex.yy.c
% cc -o qw y.tab.c lex.yy.c    # compile and link C files
```

After doing this, "qw" can be used to check an input file in the form of:

```
% qw filename
```

where the filename is the name of the specification of a contract. If there is a mistake in the Contracts specification, "qw" returns an error message, the current line number and the current token. We used "qw" to check three examples from Helm's paper [Helm 90]. These three examples are contracts DepthFirst, Dft-Connected, and Dft-Cycle, we also used "qw" to check the example in the next chapter.

Chapter 5

Example

This chapter introduces how Contracts can be used to express the behavioral composition of the participants in a mine drainage control system problem [Barb 94]. The mine drainage control system is illustrated in Fig. 5. The system involves four components, namely, the pump, pump controller, level detector, and environment monitor. They cooperate to work as follows. The pump is used to pump water at the surface mine from the sump when the water reaches a certain level. It is activated or stopped by the pump controller. The level detector is responsible for monitoring the level of water in the sump. It sends the "High" signal to the pump controller when the water accumulates to a certain level. It transmits the "Low" signal when the sump is almost empty. In the meantime, the environment monitor tests the concentration of the methane. It sends the "Alarm" signal to pump controller when the concentration of the methane reaches a certain critical level. The "Safe" signal is sent to pump controller when the concentration of methane returns to an acceptable level. The pump controller is responsible for starting and stopping the pump with the "Start" and "Stop" signals. The signals transmitted between the components is illustrated in Fig. 6. The behaviour composition of the system is described by the following MineDrainage contract.

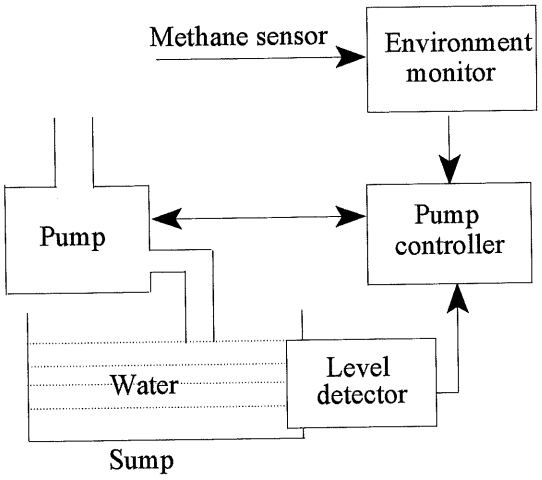

Figure 5: Computer-controlled mine drainage system

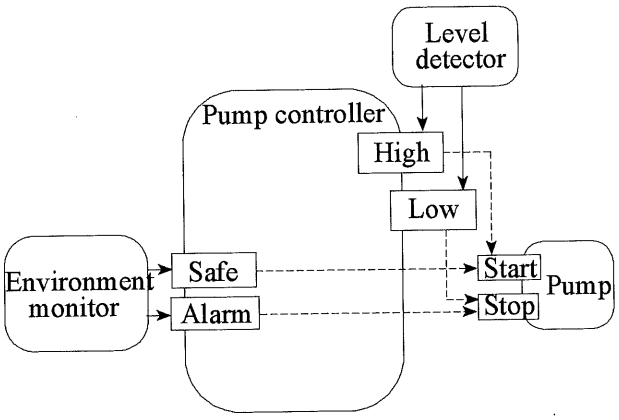

Figure 6: Signals transmitted between the components

contract MineDrainage

participants

pump_controller: PumpController

pump: Pump;

environment_monitor: EnvironmentMonitor;

level_detector: LevelDetector;

PumpController **supports**

[

LDstate: high, low; /* state of the level detector */

EMstate: safe, alarm; /* state of the environment monitor */

Pumpstate: work, idle; /* state of the pump*/

ReadPumpstate( ): [Pumpstate = pump → GetPumpstate( )];

ReadEMstate( ): [EMstate = environment_monitor →GetEMstate( )];

ReadLDstate( ): [LDstate = leveldetector → GetLDstate( )];

PumpControllerAct( ):

[ReadPumpstate( );

ReadEMstate( );

ReadLDstate( );

if Pumpstate = work **then**

{ **if** (EMstate = alarm **or** LDstate = low) **then** { pump → Stop( );}

};

if Pumpstate = idle **then**

{ **if** (LDstate = high **and** EMstate = safe) **then** { pump → Start( ); }

};

PumpControllerAct( );]

]

Pump **supports**

[state : work, idle;
GetPumpstate( ) : state [**return** state];
Start( ): [state = work];
Stop( ) : [state = idle];
]

EnviromentMonitor **supports**

[state : alarm, safe;
GetEMstate( ) : state [**return** state];
UpdateEMstate(st:state) : [state = st];
]

Level Detector **supports**

[state : high, low;
GetLDstate( ) :state [**return** state];
UpdateLDtstate(st: state): state [state = st];
]

invariant

Pumpstate=work <=> EMstate = safe **and** LDstate = high;

instantiation

PumpController → PumpControllerAct( );

endcontract

The above contract has four participants: pump_controller, pump, environment_monitor, and level_detector. The signals Start and Stop are modelled as methods on the pump participant. The signals Safe and Alarm are modelled as state changes of the environment monitor. Similarly, the signals High and Low are modelled as state changes of the level detector.

The participant pump_controller is responsible for starting or stopping the pump according to the present states of the pump, environment, and water level. So the pump controller supports the obligations of getting the present states of the pump, environment, and level detector. The method PumpControllerAct activates or stops the pump according to the three states that it gets.

The participant pump is required to support the variable state to express the state of the pump. The methods GetPumpstate( ) is used to get the state of the pump, Start( ) to start the pump, and Stop( ) to stop the pump.

The participant environment_monitor needs to support the variable state which reflects the state of the environment monitor and the methods GetEMstate( ) and UpdateEMstate( ) to get and update the state of the environment monitor.

The participant level_detector needs to support the variable state and the methods GetLDstate( ) and UpdateLDstate( ), to get and update the state of the Level Detector.

This contract implies the invariant that "If the pump is working, the present state of the environment monitor must be safe and the present state of the level detector must be high".

Chapter 6

Conclusions

Contracts generalizes the initial concept of contract for two entities to multi-object relations. This thesis has introduced the formalization of the static semantics of Contracts using attribute grammars. The syntax of Contracts used was introduced in [Holl 93].

The initial motivations for the contract approach grew out of the problems experienced when understanding and reusing large complex class libraries. Contracts focuses on the problem of understanding the important behavioral dependencies which result from object interactions. Object interactions expressed by Contracts provide a means to describe and specify the relationships and factor some of the complexity.

Contracts was used by many researchers in their work. It was used as an intermediate description technique for micro-architectures and frameworks. Micro-architectures comprise both the design and code of the classes involved and also the interactions and control flow among those classes. A framework is the design of a set of classes that collaborate to carry out a set of responsibilities.

Contracts provides the properties of flexibility, and easier reading and understanding of specifications, because it supports both the imperative and declarative styles of description.

The earliest uses of Contracts were its application in the domains of business/MIS and GUI frameworks, and the representation of variations of the classic depth first traversal algorithm as large-grained reusable object-oriented abstractions.

Recent work of Makungu and Barbeau adapted Contracts to the specification of telecommunications protocols [Maku 94]. Since the original dynamic semantics for each construct of Contracts is designed by using a denotational style method, the computation model behind contracts is sequential. Because specifying distributed routing protocols requires a parallel model of computation, their work also includes the formalization of Contracts by a parallel model of computation based on colored Petri nets [Jens 92]. They discuss modeling of objects by means of colored tokens, explain how control states of objects in Contracts are mapped to CP-nets places and express the semantics of specifications in Contracts by means of tokens, places, and transitions of CP-nets. This leads to the application of Contracts to distributed domain applications.

Based on the extensions of the Contracts approach to a parallel model of computation, work on verification can be done as future research. Verification of specifications consists of verification of general properties and verification of specific properties. Approaches to verification of specifications have followed two main paths: program proving and reachability analysis. Reachability analysis is based on constructing a reachability graph [Boch 78]. It is useful for both automatic verifications of self-consistency and comparison of two specifications at different abstraction levels. In the latter case, reachability analysis is performed on both specifications and the derived graphs are compared to determine whether or not they are equivalent in some precisely defined relation. Different relations between two specifications capture different aspects of the specifications [Pehr 89]. An often used relation is the observation equivalence [Park 81]. Observation equivalence between two processes is proven by the existence of a bisimulation relation between the state sets of the processes. There are methods and algorithms for the verification of bisimulation [Pehr 89].

Contracts can be used as a high-level specification language for the requirements specification of problems. Since reachability analysis methods for CP-nets already exist,

verification of general properties of Contracts specifications can be done. Also, comparison of specifications from two different levels of abstraction is possible. Conformance can be determined using a bisimulation relation between the reachability graphs of two CP-nets, that is, the one derived from a Contracts specification and the one derived from another lower level specification language, which also needs to be formalized by CP-nets. Thus, the requirements specification and the design specification of the same problem described by two languages at different abstraction levels can be compared.

The last things that need to be mentioned about the Contracts language are the techniques of contract inclusion and contract refinement. These provide complete and flexible mechanisms to allow contracts to be defined incrementally. These two mechanisms are analogous to the inheritance and composition mechanisms of object-oriented languages. Refinement is the analogue of inheritance and inclusion is the analogue of composition. These enable the contract designer to create new contract definitions from existing ones. Refinement supports the representation of an object interaction as a specialization of a known interaction. The designer can add new participants or enhance the obligations of an existing participant. Inclusion allows the designer to include a known interaction as part of a new more complicated interaction. With these features, the Contracts language becomes a powerful tool for design. The emphasis of design is interactions between objects, which is called Interaction-Oriented design. Contrary to class-based design, with interaction-oriented design the specification of a class becomes spread over a number of contracts and conformance declarations. It is not localized to one class definition.

Our formalization is done using attribute grammars. Compared with other definition techniques, an attribute grammar defines each language construct only by its *immediate environment*, minimizing the interconnection between the different parts of the language. This localization and partitioning of the semantic rules tend to make the definition easier to understand and more concise.

The contribution of this thesis is providing a formalization of the static semantics of Contracts language. This formal specification can be used in the case that Contracts is used to solve problems such as understanding the important behavioural dependencies which result from object interactions. Since the work in this thesis concerns only the static semantics of the Contracts, future work could be done about the dynamic semantics of the language.

Appendix A

Lex Specification of Contracts

```
%{
#include <stdio.h>
#include "y.tab.h"
int lineno=1;
%}
%%
[0-9]+ {return (INTNUM);}
[0-9]*\.[0-9]+ {return (REALNUM);}
any     {return(ANY);}
all     {return (ALL);}
and     {return (AND);}
belong   {return (BELONG);}
boolean {return (BOOLEAN);}
contract {return (CONTRACT);}
do      {return (DO);}
end     {return (END);}
exist    {return (EXIST);}
false   {return (FALSE);}
for     {return (FOR);}
forany  {return (FORANY);}
if      {return (IF);}
in      {return (IN);}
includes {return (INCLUDES);}
instantiation {return (INSTANTIATION);}
integer       {return (INTEGER);}
invariants    {return (INVARIANTS);}
not     {return (NOT);}
of      {return (OF);}
or      {return (OR);}
participants {return (PARTICIPANTS);}
real        {return (REAL);}
refines {return (REFINES);}
return  {return (RETURN);}
set     {return (SET);}
supports {return (SUPPORTS);}
then    {return (THEN);}
true    {return (TRUE);}
void    {return (VOID);}
while   {return (WHILE);}
```

```
[a-z][a-zA-Z0-9]*_?[a-zA-Z0-9]* {return (LIDENTIFIER);}
[A-Z][a-zA-Z0-9]*_?[a-zA-Z0_9]* {return (UIDENTIFIER);}
[ \t]+ ;
"<" |
">" |
"==" |
">=" |
"<=" |
"<>"  {return (COMPARISON);}
"+" |
"-" |
"*" |
"/"   {return (ARITHMETIC);}
"=" |
"(" |
")"  {return yytext[0];}
"=>" |
"<=>"  {return (LOGIC);}
"->"   {return (CALL);}
"|->"  {return (LEADTO);}
\n lineno++;
. {return yytext[0];}
%%
void yyerror(s)
char *s;
{ fprintf(stderr, " %s\n", s);
  printf("line%d:     %s at %s\n", lineno, s, yytext);}
main(argc, argv)
int argc;
char **argv;
{
        if (argc>1) {
                FILE *file;
                file = fopen(argv[1], "r");
                if (!file) {
                        fprintf(stderr, "could not open %s\n", argv[1]);
                        exit(1);
                          }
                yyin=file; }
                if (!yyparse())
                        printf("parse worked\n");
                else
                        printf("parse failed\n");
}
```

```
yywrap()
{
 return(1);
}
```

Appendix B

Yacc Grammar of Contracts

```
%start contracts
%token ALL ANY BELONG BOOLEAN CALL CONTRACT DO END EXIST FALSE
%token FOR FORANY IF IN INCLUDES THEN LIDENTIFIER INTEGER INTNUM
%token INSTANTIATION INVARIANTS PARTICIPANTS OF REAL REALNUM
%token REFINES RETURN SET SUPPORTS TRUE UIDENTIFIER VOID WHILE
%left   '.' '='  COMPARISON ARITHMETIC AND LEADTO LOGIC OR NOT
%%
contracts:
        contract
        |
        contracts ';' contract
        ;
contract:
        CONTRACT IDENTIFIER
        PARTICIPANTS  participant_declarations
        inclusion_part
        refinement_part
        obligation_specifications
        instantiation
        invariants
        END CONTRACT
        ;
IDENTIFIER:
        LIDENTIFIER
        |
        UIDENTIFIER
        ;
participant_declarations:
        participant_declarations  ';' participant_declaration
        |
        participant_declaration
        ;
participant_declaration:
        LIDENTIFIER ':' participant_type_specification
        ;
participant_type_specification:
        UIDENTIFIER
        |
        SET '(' UIDENTIFIER ')'
        ;
```

```
inclusion_part:
        |
        inclusions';'
        ;
inclusions:
        inclusions ';' inclusion
        |
        inclusion
        ;
inclusion:
        INCLUDES IDENTIFIER
        |
        INCLUDES IDENTIFIER '(' expressions ')'
        ;
refinement_part:
        |
        refinements';'
        ;
refinements:
        refinements ';'refinement
        |
        refinement
        ;
refinement:
        REFINES IDENTIFIER
        |
        REFINES IDENTIFIER '(' expressions ')'
        ;
obligation_specifications:
        obligation_specifications  obligation_specification
        |
        obligation_specification
        ;
obligation_specification:
        UIDENTIFIER SUPPORTS  '['obligations']'
        ;
obligations:
        variable_declarations';' methods
        |
        methods
        ;
variable_declarations:
        variable_declarations ';' variable_declaration
        |
        variable_declaration
        ;
```

```
variable_declaration:
        IDENTIFIER ':' type_specification
        ;
methods:
        methods';'method
        |
        method
        ;
method:
        IDENTIFIER '(' parameter_type_list')' ':'rtype_specification
        |
        IDENTIFIER '(' parameter_type_list')' ':'rtype_specification '['statements']'
        |
        IDENTIFIER '('parameter_type_list')' ':' '['statements']'
        ;

parameter_type_list:
        |
        parameter_declarations
        ;
parameter_declarations:
        parameter_declarations ',' parameter_declaration
        |
        parameter_declaration
        ;
parameter_declaration:
        LIDENTIFIER ':' type_specification
        ;

rtype_specification:
        type_specification
        |
        VOID
        ;
type_specification:
        BOOLEAN
        |
        INTEGER
        |
        REAL
        |
        participant_type_specification
        |
        LIDENTIFIER
        ;
```

```
instantiation:
          |
        INSTANTIATION statements
          ;
invariants:
          |
         INVARIANTS expressions ';'
          ;
statements:
        statements ';' statement
          |
       statement
          ;
statement:
          |
        assignment
          |
        IF expression THEN '{' statements '}'
          |
        FOR ALL LIDENTIFIER IN type_specification '{' statements '}'
          |
        condition
          |
        WHILE expression DO '{' statements '}'
          |
         RETURN expression
          |
         name CALL method_call
          |
          method_call
          ;
condition:
         '{' expression'}'
          ;
expression:
         expression COMPARISON expression
          |
         name
          |
         number
          |
         TRUE
          |
         FALSE
          |
         '(' expression ')'
```

```
        |
        NOT '('expression')'
        |
        expression ARITHMETIC expression
        |
        expression AND expression
        |
        expression OR expression
        |
        expression LOGIC expression
        |
        name CALL method_call
        |
        method_call
        |
        expression '.' expression
        |
        expression LEADTO '['statements']'
        |
        FORANY LIDENTIFIER  IN UIDENTIFIER ':' '['statements']'
        |
        EXIST LIDENTIFIER IN UIDENTIFIER ':' '['statements']'
        ;
expressions:
        |
        expressions ','expression
        |
        expression
        ;
assignment:
        name'='expression
        ;
name:
        LIDENTIFIER
        |
        UIDENTIFIER
        ;
number:
        INTNUM
        |
        REALNUM
        ;
method_call:
        IDENTIFIER '(' expressions ')'
        ;
%%
```

References:

[Alex 79] C. Alexander, *The Timeless Way of Building*, Oxford University Press, New York, NY, 1979.

[Bakk 67] J. W. De Bakker, *Formal Definition of Programming Languages*, with An Application to the Definition of ALGOL 60, Math Cent. Tracts 16, Mathematisch Centrum, Amsterdam, 1967.

[Barb 94] M. Barbeau, G. Custeau, and R. St-Denis, *Requirements Engineering and Synthesis of a Control System*, Automatic Control Production Systems, Automatique, Productique, Informatique Industrielle (APII), Vol. 28, No. 1, 1994, pp. 37-52.

[Buhr 92] R. J. A. Buhr and R. S. Casselman, *Architectures with Pictures*, OOPSLA'92 pp. 466-483.

[Boch 78] G. V. Bochmann, *Finite State Description of Communication Protocols*, Computer Networks, Vol. 2, October 1978, pp. 361-372.

[Budd 91] T. Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley, 1991.

[Cole 92] D. Coleman, F. Hayes and S. Bear. *Introducing Objectcharts or How to use Statecharts in Object-Oriented Design*, IEEE Transactions on Software Engineering, Vol. 18, No.1, Jan. 1992, pp.8-18.

[Gamm 93] E. Gamma, R. Helm, R. Johnson, and J.Vlissides, *Design Patterns: Abstraction and Reuse of Object Oriented Design.* In Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserlautern, Germany, July, 1993.

[Haye 91] F. Hayes and D. Coleman, *Coherent Models for Object-Oriented Analysis*, Proceedings of OOPSLA'91, SIGPLAN Notices, Vol. 26, No. 11, November, 1991, pp. 171-183.

[Helm 90] R. Helm, I. M. Holland, and D. Gangopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, Proceedings of ECOOP/OOPSLA'90, Ottawa 1990, pp. 169-180.

[Holl 92] I. M. Holland, *Specifying Reusable Components Using Contracts*, Proceedings of European Conference on Object-Oriented Programming 1992 (ECOOP'92), LNCS 615 Springer-Verlag, pp. 287-308.

[Holl 93] I. M. Holland, *The Design and Representation of Object-Oriented Components*, Ph.D. Thesis from Northeastern University, Boston, 1993.

[Iron 61] E. T. Irons, *A Syntax Directed Compiler for Algol 60*, Communications of the ACM 4, 1961.

[Iron 63] E. T. Irons, *Towards More Versatile Mechanical Translator*, Proceedings of Sympos. Appl. Math., Vol. 15, Amer. Math. Soc., Providence, RI, 1963, pp. 41-50.

[Jens 92] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods, and Practical Use*, Volume 1, Springer-Verlag, 1992.

[John 92] R. Johnson, *Documenting Frameworks Using Patterns*. In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications, Vancouver, B. C., October, 1992, pp. 63-76.

[Knut 68] D. E. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems

Theory, Vol. 2, No.2, June 1968, pp. 127-146.

[Lajo 94] R. Lajoie and R. K. Keller, *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert*, 62ième Congrès de l'Association Canadienne Française pour l'Avancement des Sciences (ACFAS), Montréal, Canada.

[Land 64] P. J. Landin, *The Mathematical Evaluation of Expressions* , Comp. J., 6, 1964, pp. 308-320.

[Land 65] P. J. Landin, *A Correspondance between ALGOL 60 and Church's Lambda Notation*, Comm. ACM 8, 1965, pp. 89-101.

[Land 66] P. J. Landin, *A Formal Description of ALGOL 60*, Formal Language Description Languages for Computer Programming, Proc. IFIP WORKING Conf., Vienna, North Holland, 1966, pp. 266-294.

[Levi 92] J. R. Levine, T. Mason and D. Brown, *Lex and Yacc*, O`Reilly & Associates Inc., October, 1992.

[Maku 94] M. Makungu and M. Barbeau, *The Contracts Approach : Formalization and Application to Communications Protocols*, Septièmes Entretiens du Centre Jacques Cartier, Colloque Informatique Communicante et Systèmes Distributés / Communicating Informatics and Distributed Systems, Grenoble, novembre-decembre 1994, pp. 85-102.

[Meye 92] B. Meyer, *Applying "Design by Contract"*, Computer, October 1992, pp. 40-51.

[Park 81] D. M. R. Park, *Concurrency and Automata on Infinite Sequences*, Proc of 5th GI Conf. on Theoretical Computer Science, LNCS 104, Springer-Verlag, 1981, pp. 167-183.

[Pehr 89] B. Pehrson, *Protocol Verification for OSI*, Computer Networks and ISDN Systems 18 (1989/1990), pp. 185-201.

[Rumb 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Schr 91] M. Schrefl and G. Kappel, *Cooperation Contracts*, In International Conference on the ER Approach, North-Holland, 1991.

[Wait 85] W. M. Waite and G. Goos, *Compiler Construction*, Springer-Verlag, 1985.

[Wirf 89] R. Wirfs-Brock and B. Wilkerson. *Object-oriented Design: a Responsibility-Driven Approach*, in Object-Oriented Programming Systems, Languages and Applications Conferences, pp. 71-76, ACM, New Orleans, LA 1989.

[Wirf 90] R. Wirfs-Brock and R. E. Johnson, *A Survey of Current Research in Object-Oriented Design*, Communications of the ACM, November, 1990, pp. 104-113.